Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Hasir Mesić

Predpomnilnik in indeksiranje nestrukturiranih podatkov

diplomsko delo

visokošolski strokovni študijski program prve stopnje računalništvo in informatika

Mentor: dr. Andrej Brodnik

Ljubljana 2014

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Fakulteta za računalništvo in informatiko izdaja naslednjo nalogo:

Tematika naloge:

Pri sodobni analizi podatkovnih struktur in algoritmov na njih upoštevamo pomnilniško hierarhijo. Slednja sestoji iz več stopenj, od katerih prvo predstavlja predpomnilnik. Velikost le-tega igra pomembno vlogo pri učinkovitosti podatkovnih struktur.

Literatura omenja dva v osnovi različna pristopa k načrtovanju podatkovnih struktur: takšnega, ki se ne zaveda velikosti predpomnilnika in takšnega, ki se le-tega zaveda. V diplomski nalogi primerjajte podatkovni strukturi ERA (Elastic Range) in COSD (Cache Oblivious Dictionary) kot predstavnika obeh pristopov načrtovanja ter ugotovite razlog za njuno različno kakovost.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Hasir Mesić, z vpisno številko **63090342**, sem avtor diplomskega dela z naslovom:

Predpomnilnik in indeksiranje nestrukturiranih podatkov

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom dr. Andreja Brodnika,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela na svetovnem spletu preko univerzitetnega spletnega arhiva.

V Ljubljani, dne 8. julij 2014 Podpis avtorja:

Pri pisanju diplomske naloge se zahvaljujem mentorju in njegovemu asistentu Matevžu Jekovcu za strokovno pomoč in nenehno dosegljivost.

Svojemu nečaku Arslanu.

Kazalo

Slike

Tabele

Algoritmi

Seznam uporabljenih kratic

kratica	angleško	slovensko
COSD	cache-oblivious string dictionary	predpomnilniško-nezavedni slovar nizov
DNA	deoxyribonucleic acid	deoksiribonukleinska kislina
ERA	elastic range algorithm	algoritem z dinamičnim prilagajanjem globine priponskega drevesa

Povzetek

V diplomski nalogi sta predstavljena algoritma za gradnjo podatkovnih struktur, ki hranita dolga besedila. Na začetku je predstavljena struktura molekule DNK, kot primer dolgega besedila. Opisani so različni primeri podatkovnih struktur. V istem sklopu je opisana tudi pomnilniška hierarhija, ki vpliva na hitrost izvajanja algoritmov.

V osrednjem delu je posamezen algoritem predstavljen v svojem poglavju, kjer je postopek gradnje podatkovne strukture razdeljen v več faz. Vsaka faza je podrobno opisana in ponazorjena s konkretnim primerom.

V zadnjem sklopu je podana primerjava algoritmov glede na časovno in prostorsko zahtevnost, tako za gradnjo podatkovne strukture kot za poizvedbe. Predstavljeni so tudi rezultati časovnih meritev in meritev V/I dostopov.

Ključne besede: DNK, podatkovna struktura, predpomnilnik, ERA, COSD.

Abstract

In this thesis algorithms for construction of data structures for a long texts, are introduced. In the beginning, the overview of the DNA structure is given, as an example of a long text. Several examples of data structures are described. In the same part of thesis, the memory hierarchy, which influences algorithm execution speed, is described.

In the main part each algorithm is presented in its own chapter, where the construction process is divided into number of stages. Each stage is described and illustrated with concrete example.

The last part gives the comparison of algorithms with respect to time and space complexity, both for the construction of data structures as queries. It also presents the results of time measurements and measurements of I/O accesses.

Keywords: DNA, data structure, cache, ERA, COSD.

Poglavje 1

Uvod

Od trenutka, ko je človek ukrotil ogenj, si nenehno prizadeva k spreminjanju načina življenja in svoje okolice ter stremi k odkrivanju novih fizikalnih in naravnih zakonitosti, ki so vse bolj zapletena in v območju nevidnega. Mnoga spoznanja ne bi bila mogoča brez uporabe zmogljivih računalniških kapacitet. Eno izmed takih spoznanj se nanaša na področje genetike, ki preučuje dedovanje, lastnosti genov in DNK. Slednja je sestavljena iz genov, ki se nahajajo v kromosomih.

Postopek sestavljanja človeškega genoma je večopravilen, saj poteka na večjedrnih računalnikih. Izziv je shraniti DNK v podatkovno strukturo, ki uporablja manj prostora in hkrati omogoča čim cenejše, večkratno, iskanje njenih posameznih delov. Glede na dostopno literaturo in razvite algoritme ni bilo najdenih, za oba izziva hkrati, ustreznih rešitev.

V diplomski nalogi bom predstavil in primerjal dva algoritma. Prvi rešuje izziv prostorske racionalnosti, drugi pa teoretično omogoča časovno učinkovite operacije iskanja.

1.1 DNK

Fosilni ostanki kamnin starih tri in pol milijarde let kažejo na znake biološkega kopičenja dušika prvih bakterij. Do danes so se organizmi razvili od enoceličnih, kot so bakterije, do večceličnih, med katere sodijo rastline, živali in ljudje. Prilagodili so se povsod tam, kjer jim naravno okolje omogoča pridobivanje energije za življenje. Tako jih najdemo v vodi, zraku, kopnem, zemlji in kamninah. Zmožni

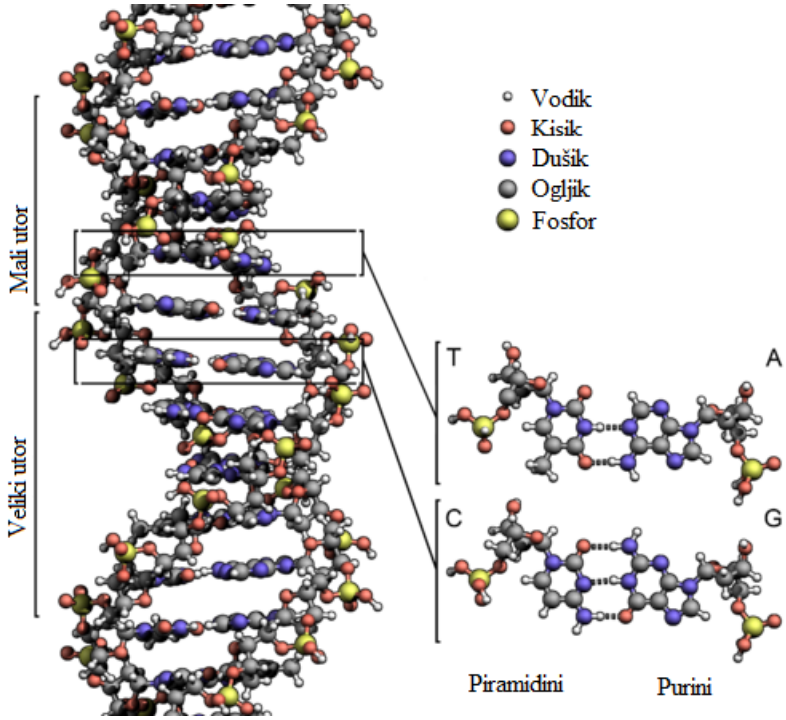
so preživeti v pogojih, kjer temperature dosegajo skoraj vrelišče in kjer se temperature spustijo pod ledišče vode [1].

Organizme delimo v tri velike skupine: bakterije, arhaeje (bakterije znane po tem, da preživijo v ekstremnih pogojih) in evkariote. Prvi dve skupini združujemo v tako imenovane prokariote. Bistvena razlika med prokarioti in evkarioti je, da prvi sodijo v skupino enoceličnih organizmov in se po strukturi in organizaciji celice razlikujejo po tem, da so manjše in manj kompleksne, njihova DNK pa od citoplazme ni ločena z jedrno membrano. Odsotni sta tudi organeli - kompleksne celične membrane, kot sta mitohondrij in kloroplast. Delitev celic pri prokariotih poteka mnogokrat hitreje kot pri evkariotih [1].

Vsi organizmi si delijo skupne značilnosti [1]:

- osnovna enota življenja je celica,
- kemijska energija se shranjuje v ATF (Adenozin-trifosfat),
- genetske informacije so shranjene v DNK,
- informacije se prepisujejo v RNK (ribonukleinska kislina),
- obstaja skupni trojček genetske kode (z nekaterimi izjemami),
- prevod informacij v proteine vključuje ribosome,
- skupni biokemični procesi (glikoliza, podvajanje in popravljanje DNK,...) in
- podobni proteini so zelo razširjeni med različnimi skupinami organizmov.

Molekula DNK je nosilka genetske informacije vseh živih organizmov. Osnovni gradnik molekule DNK je nukleotid, ki je sestavljen iz sladkorja (deoksiriboza), dušikove baze (adenin, citozin, gvanin in timin) in fosfatne skupine. Štirje različni nukleotidi tvorijo genetsko abecedo življenja na Zemlji, ki jih označujemo z velikimi latinskimi črkami A, C, G in T. Zaporedje nukleotidov pa, podobno kot zaporedje črk v besedi, določa pomen genetske informacije. V vseh živih organizmih (z izjemo nekaterih virusov) ima DNK obliko dvojne vijačnice, pri čemer se dve molekuli DNK ovijeta druga okrog druge, kot to prikazuje Slika 1.1 [9]. Pri tem se dušikove baze nahajajo znotraj vijačnice in se medsebojno uparijo. Adenin se vedno upari s timinom in citozin vedno z gvaninom (Watson-Crickovo pravilo baznih parov) [1]. Celotno zaporedje baznih parov nukleotidov oziroma DNK zaporedje, ki se nahaja v 23 parih kromosomov celičnega jedra (22 avtosomov in en par spolnih kromosomov) in mitohondrijska DNK, tvori človeški genom, ki obsega



Slika 1.1: Struktura DNK.

približno tri milijarde DNK baznih parov [9].

1.2 Struktura naloge

V prvem poglavju je predstavljen namen diplomske naloge, osnove DNK in struktura diplomskega dela. Sledi opis osnovnih značilnosti podatkovnih struktur, ki hranijo besedila, in pomnilniške hierarhije. Delovanje algoritma ERA za gradnjo priponskih dreves je opisano v tretjem poglavju. Sledi poglavje o delovanju algoritma COSD. V petem poglavju so predstavljene primerjave rezultatov časovnih meritev in meritev V/I dostopov poizvedb za predstavljena algoritma. Sledijo sklepne ugotovitve. V dodatku so dodani opisni deli algoritma ERA ter tabele, ki prikazujejo rezultate meritev V/I dostopov poizvedb za več testnih primerov.

Poglavje 2

Osnove

V tem poglavju predstavljamo osnove podatkovnih struktur, ki so uporabljene v diplomskem delu ter model arhitekture računalnika, ki smo ga uporabili pri študiji in primerjavi rezultatov.

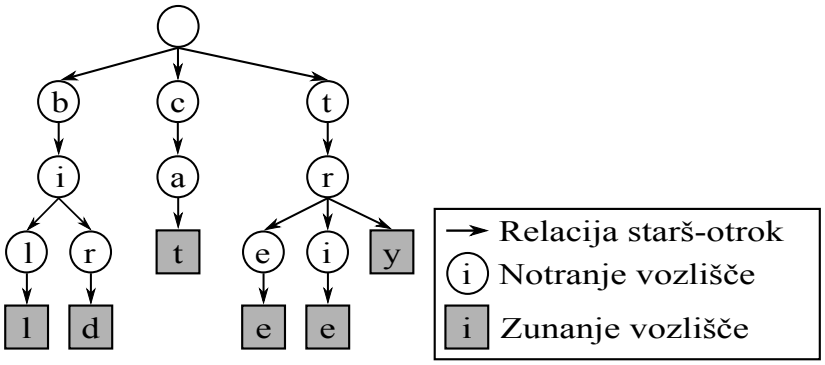
2.1 Podatkovne strukture

Podatkovno strukturo v računalništvu opredeljujemo kot način hranjenja in organizacije podatkov v pomnilniku, ki določa predvsem učinkovitost izvajanja algoritmov, s katerimi določene podatke iščemo, jih spreminjamo ali brišemo. Pri izbiri podatkovne strukture moramo biti previdni in vedno upoštevati, katere operacije nad podatki bodo najbolj pogosto uporabljene. S tem sprejmemo kompromis, da bo izvajanje nekaterih operacij hitrejše od ostalih. Kot primer naj navedemo hranjene urejenih podatkov, ki lahko pohitrijo iskanje podatka na račun počasnejšega vstavljanja; in obratno, hranjenje neurejenih podatkov omogoča hitrejše vstavljanje in počasnejše iskanje podatka [3].

V nadaljevanju so predstavljene podatkovne strukture: številsko drevo, številsko drevo s stiskanjem poti in priponsko drevo, ki so namenjene predvsem hranjenju množice besed. Zadnji dve drevesi predstavljata optimizirani in prilagojeni verziji splošnega številskega drevesa.

2.1.1 Številsko drevo

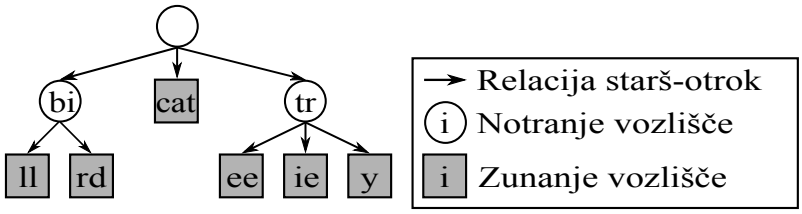
Številsko drevo (ang. *trie*) je podatkovna struktura, ki hrani množico besed, katere namen je zagotavljanje hitrega iskanja besed. Vsa vozlišča, razen korena drevesa, so označena s črko iz končne abecede Σ, nasledniki vozlišč pa so urejeni po abecedi. Povezave od korena do zunanjih vozlišč, obarvana s sivo, tvorijo besede prvotno podane množice besed [4]. Slika 2.1 prikazuje primer številskega drevesa za besede *"bill", "bird", "cat", "tree", "trie" in "try"*.



Slika 2.1: Številsko drevo za besede *"bill", "bird", "cat", "tree", "trie" in "try"*.

2.1.2 Številsko drevo s stiskanjem poti

Številsko drevo s stiskanjem poti (ang. *PATRICIA - Practical Algorithm To Retrieve Information Coded In Alphanumeric*) je v osnovi splošno številsko drevo, ki zagotavlja, da ima vsako notranje vozlišče drevesa vsaj dva naslednika. To je zagotovljeno tako, da vsa vozlišča z enim naslednikom združi v eno povezavo. Prednost takih dreves je v tem, da je število vozlišč sorazmerno številu besed in ne skupni dolžini množice besed [4]. Na Sliki 2.2 je prikazan primer številskega drevesa s stiskanjem poti z istimi besedami kot v Poglavju 2.1.1.

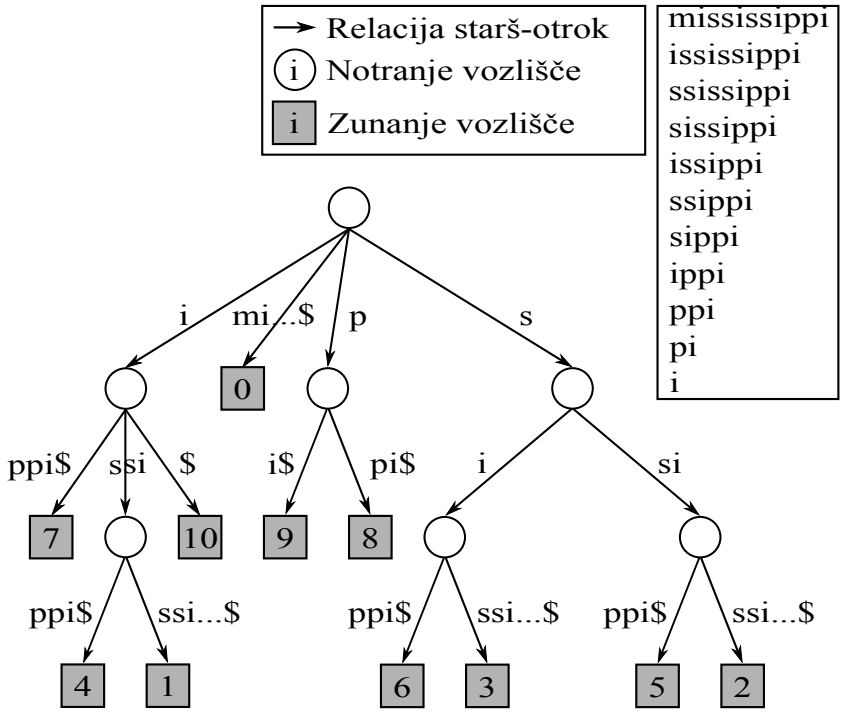


Slika 2.2: Številsko drevo s stiskanjem poti za besede *"bill", "bird", "cat", "tree", "trie" in "try"*.

2.1.3 Priponsko drevo

Priponsko drevo (ang. *Suffix tree*) omogoča hitro in učinkovito iskanje ter primerjanje priponskih besed podanega besedila. Tako omenjena struktura mora hraniti vse pripone podanega besedila. Priponska drevesa so primerna za hranjenje dolgih besedil, ki se ne spreminjajo in s tem omogočajo hitro iskanje priponskih besed [4]. Končna vozlišča hranijo indekse, na katerem mestu se, v vstavljenem besedilu, začne pripona.

Slika 2.3 prikazuje priponsko drevo, ki je zgrajeno iz besede *"mississippi"*. Drevo se zgradi tako, da v drevo vstavljamo vsako priponsko besedo posebej. Na koncu besede, ki jo vstavljamo, dodamo poseben znak $, ki ni med ostalimi črkami in označuje konec prvotne besede.



Slika 2.3: Priponsko drevo za besedo *"mississippi"*.

2.2 Pomnilniška hierarhija

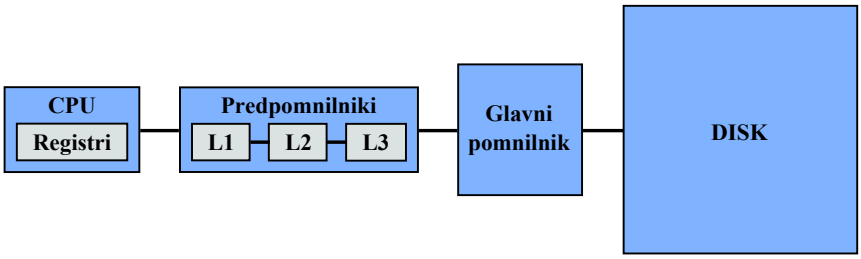
2.2.1 Splošne značilnosti

Pomnilniški sistem sodobnega računalnika se deli na več nivojev. Vsak nivo pomnilnika deluje kot predpomnilnik naslednjemu nivoju. Pomnilniki, ki se nahajajo bližje CPU so manjši, hitrejši in dražji. S povečevanjem razdalje od CPU se pomnilnikom povečuje velikost v bajtih, niža se jim cena in povečuje časovni dostop do naslednjega nivoja [6].

Slika 2.4 prikazuje dvo-nivojsko pomnilniško arhitekturo, kjer se prvi nivo deli na CPU registre, predpomnilnike in glavni pomnilnik, medtem ko drugi nivo predstavlja sekundarne pomnilnike z lokalnim (lokalni disk) ali oddaljenim dostopom (kasete, porazdeljeni datotečni sistemi, spletni strežniki). Čas dostopa do registrov, predpomnilnikov in glavnega pomnilnika se meri v nano sekundah (ns), in sicer 0.25 ns, 1.0 ns in 100.0 ns. Čas dostopa do diska se dramatično poveča na 5

mili sekund [10].

Predpomnilniška hierarhija lahko vsebuje enega ali več predpomnilnikov. Na Sliki 2.4 smo jo predstavili kot tri-nivojsko, ker je kot taka implementirana na računalniku, ki ga bomo uporabljali za primerjavo algoritmov ERA in COSD. Tehnične podrobnosti računalnika so prikazane v Tabeli 5.1.



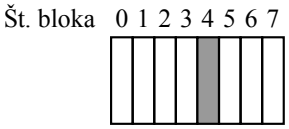
Slika 2.4: Dvo-nivojska pomnilniška arhitektura.

Podatki med pomnilnikom in diskom se prenašajo v blokih, ki je sestavljen iz več pomnilniških besed. Na enak način poteka prenos podatkov med glavnim pomnilnikom in predpomnilnikom. Med predpomnilnikom in registrom se prenašajo podatki, ki so velikosti pomnilniške besede [10].

2.2.2 Načini zapisovanja blokov v predpomnilnik

Obstajajo trije načini zapisovanja blokov v predpomnilnik, in sicer direktno naslavljanje, čisto asociativno naslavljanje in set-asociativno naslavljanje [10]. Primer direktnega naslavljanja bloka z naslovom 12 v 8-blokovni predpomnilnik prikazuje Slika 2.5. To je storjeno s preprostim izračunom:

številka bloka = (naslov bloka) mod (število blokov v predpomnilniku)

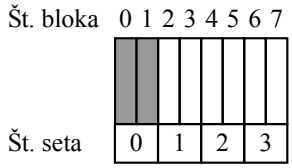


Slika 2.5: Direktno naslavljanje bloka v predpomnilnik.

Pri čistem asociativnem zapisovanju se posamezen blok zapiše na katerokoli mesto v predpomnilniku. Pri set-asociativnem zapisovanju je predpomnilnik razdeljen na različne velikosti setov blokov, ki morajo biti deljivi s številom blokov v predpomnilniku. V praksi se uporabljajo seti velikosti: 2, 4 ali 8. V kateri set se bo zapisal posamezen blok, se izračuna na sledeč način:

številka bloka = (naslov bloka) mod (število setov v predpomnilniku)

Slika 2.6 prikazuje primer set-asociativnega naslavljanja bloka z naslovom 12 v 8-blokovni predpomnilnik s stopnjo asociativnosti 2.



Slika 2.6: Set-asociativno naslavljanje bloka v predpomnilnik s stopnjo asociativnosti 2.

2.2.3 Strategije zamenjave blokov predpomnilnika

Za prepis pomnilniške besede iz predpomnilnika v register skrbi prevajalnik, iz glavnega pomnilnika v predpomnilnik pa strojna oprema. Za prepis bloka z diska v predpomnilnik poskrbi operacijski sistem, kateremu je prepuščena tudi odločitev, kateri blok bo zamenjan. V primerih direktnega naslavljanja blokov take odločitve ne obstajajo, zato se strategije zamenjave blokov nanašajo predvsem na primere, ko gre za čisto ali set-asociativno naslavljanje. Obstajajo tri strategije zamenjav blokov [10]:

- naključno izbran blok,
- nazadnje uporabljen blok (ang. *Least-recently used*) in
- najstarejši blok (ang. *First in, First out*)

V večini primerov, kot je razvidno zgoraj, nimamo vpliva na to, kateri del podatkov se prenese bližje pomnilniku in v splošnem tudi ne na to, koliko se jih prenese. Zato je razumljivo, da se poskuša načrtovati podatkovne strukture, ki se ne zavedajo predpomnilniške arhitekture in so hkrati optimalne.

Poglavje 3

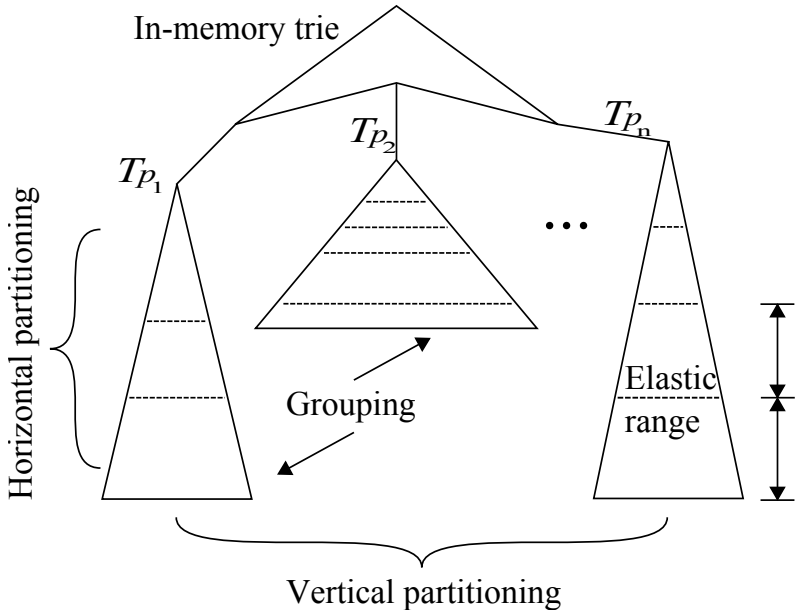
ERA

3.1 Osnovne značilnosti

Celotno poglavje je namenjeno predvsem opisu delovanja algoritma ERA [5] za gradnjo priponskih dreves, ki učinkovito deluje za zelo dolga besedila, in jih ni mogoče shraniti v glavni pomnilnik. Tako so vsi podatki priponskega drevesa shranjeni na lokalnem disku. Algoritem ERA gradi priponsko drevo v dveh fazah: vertikalno in horizontalno. Na ta način zmanjša rabo vzhodno-izhodnih naprav. To stori tako, da dinamično prilagaja horizontalne dele neodvisno od vertikalnih delov. Kadar je to mogoče, med gradnjo priponskega drevesa poskrbi tudi za združevanje posameznih poddreves, in sicer z namenom amortiziranja rabe vzhodno-izhodnih naprav. Deluje tako na eno-procesorskih kot več-procesorskih sistemih. Slika 3.1 prikazuje, kako je gradnja priponskega drevesa razdeljena na manjše podprobleme.

V vertikalni fazi je priponsko drevo razdeljeno na priponska poddrevesa T_{p_1}, T_{p_2}, ..., T_{p_n}. Na število priponskih poddreves vplivamo z določitvijo njihove maksimalne velikosti v bajtih. Nekatera priponska poddrevesa je možno združiti v skupine, saj njihova skupna velikost ne presega oziroma je enaka maksimalno določeni velikosti. Z združevanjem priponskih poddreves v skupine učinkovito izrabimo delovanje vseh jeder v več-jedrnih sistemih.

V horizontalni fazi poteka gradnja priponskih poddreves. Pred tem algoritem ERA pripravi podatke o indeksih, kje v besedilu se nahajajo pripone korena poddrevesa, in podatke o tem, katere pripone si delijo skupne predpone.



Slika 3.1: Vertikalna in horizontalna delitev gradnje priponskega drevesa.

3.2 Vertikalna delitev

Prva faza je vertikalna delitev priponskega drevesa, v kateri Algoritem 1 razdeli drevo na priponska poddrevesa $T_{p_1}, T_{p_2}, \ldots, T_{p_n}$ s predponami $p_1, p_2, \ldots, p_n$. Velikost T_p mora biti manjša ali enaka velikosti rezerviranega pomnilnika. Ker ima uporabnik neposreden vpliv na določanje velikosti priponskih poddreves, spada algoritem v predpomnilniško zavedne algoritme, saj v naprej razpolaga z informacijo o velikosti razpoložljivega pomnilnika. Vertikalna faza dejansko poskrbi, da se bodo vsi podatki v fazi gradnje poddrevesa nahajali v rezerviranem pomnilniku in ne na disku. Na ta način se učinkovito zmanjša raba vzhodno-izhodnih naprav.

Kolikšno bo število T_p, je odvisno od števila priponskih besed f_p, ki se začnejo s predpono p. Priponsko poddrevo T_p je lahko shranjeno v pomnilnik, če je $f_p \leq F_M$, kjer je

$$F_M = \frac{MTS}{2 * velikost\ vozlišča}$$

in MTS pomeni velikost rezerviranega pomnilnika za T_p.

Algoritem 1 najprej ustvari povezan seznam s predponami P iz začetnih črk abecede Σ. V zanki prebere vstavljeno besedo in za vsako predpono v seznamu P

prešteje število ponovitev f_p.

Če za trenutno predpono velja $f_p > F_M$, jo odstrani iz P. Odstranjeno predpono razširi z vsako črko iz Σ. Število novih predpon je enako velikosti $|\Sigma|$, ki se jih doda v P. Če je $f_p \leq F_M$, predpono odstrani iz P in doda v končni seznam predpon P'. To se ponavlja toliko časa, dokler obstaja predpona v P. Nato Algoritem 1 uredi P' po velikosti glede na f_p. V zadnji zanki, dokler obstajajo predpone v P', gradi skupine, tako da sešteva f_p priponskih besed. Če je $\sum_{i=0}^{|P'|} f_{p_i} \leq F_M$, doda predpono v skupino in skupino v množico poddreves.

Za predstavitev in ponazoritev gradnje priponskega drevesa izberimo niz S = TGGTGGTGGTGCGGTGATGGTGC, ki je zaključen s posebnim znakom \$, abecedo Σ = {A, C, G, T} in $F_M = 5$. V nadaljevanju je na konkretnem primeru, korak za korakom, predstavljeno delovanje Algoritma 1, ki na začetku nastavi navidezno drevo predpon $V_T = \{\}$, seznam P = {A, C, G, T} in seznam $P' = \{\}$. Tabela 3.1 prikazuje delovanje prvega dela Algoritma 1, kjer se po korakih spreminjata P in P'. Vrednosti pod stolpcem $f_p \leq F_M$ pomenita t - pogoj drži in f - pogoj ne drži .

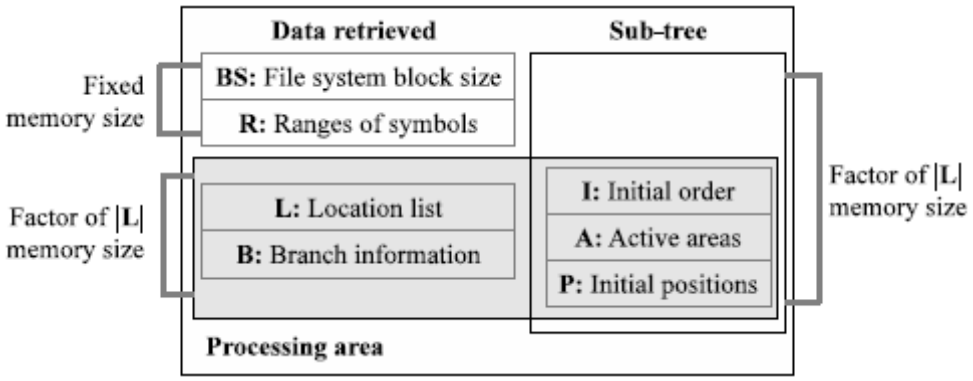
i	p	f_p	$f_p \leq F_M$	P	P'
1	A	1	t	{C,G,T}	{A}
2	C	2	t	{G,T}	{A,C}
3	G	13	f	{T,GA,GC,GG,GT}	{A,C}
4	T	7	f	{GA,GC,GG,GT,TA,TC,TG,TT}	{A,C}
5	GA	1	t	{GC,GG,GT,TA,TC,TG,TT}	{A,C,GA}
6	GC	2	t	{GG,GT,TA,TC,TG,TT}	{A,C,GA,GC}
7	GG	5	t	{GT,TA,TC,TG,TT}	{A,C,GA,GC,GG}
8	GT	5	t	{TA,TC,TG,TT}	{A,C,GA,GC,GG,GT}
9	TA	0	f	{TC,TG,TT}	{A,C,GA,GC,GG,GT}
10	TC	0	f	{TG,TT}	{A,C,GA,GC,GG,GT}
11	TG	7	f	{TT,TGA,TGC,TGG,TGT}	{A,C,GA,GC,GG,GT}
12	TT	0	f	{TGA,TGC,TGG,TGT}	{A,C,GA,GC,GG,GT}
13	TGA	1	t	{TGC,TGG,TGT}	{A,C,GA,GC,GG,GT,TGA}
14	TGC	2	t	{TGG,TGT}	{A,C,GA,GC,GG,GT,TGA,TGC}
15	TGG	4	t	{TGT}	{A,C,GA,GC,GG,GT,TGA,TGC,TGG}
16	TGT	0	f	{}	{A,C,GA,GC,GG,GT,TGA,TGC,TGG}

Tabela 3.1: Generiranje predpon.

Končni seznam predpon P' se v drugem delu Algoritma 1 uredi padajoče glede na f_p. Tako iz prvotnega seznama $P' = \{(A,1), (C,2), (GA,1), (GC,2), (GG,5), (GT,5), (TGA,1), (TGC,2), (TGG,4)\}$ dobimo po velikosti urejen seznam $P' = \{(GG,5), (GT,5), (TGG,4), (C,2), (GC,2), (TGC,2), (A,1), (GA,1), (TGA,1)\}$. Seznam P' vsebuje 9 predpon, ki dejansko predstavljajo priponska poddrevesa. Element seznama P' je v obliki terke, kjer ključ predstavlja predpono p, vrednost ključa pa f_p. Ker so nekatere vrednosti terk manjše od meje F_M, jih je smiselno združiti v skupine, če je $\sum_{i=0}^{|P'|} f_{p_i} \leq F_M$, za kar poskrbi tretji in zadnji del Algoritma 1. Glede na dobljen seznam P', se predpone združijo v 5 skupin in se dodajo v navidezno drevo predpon $V_T = \{\{(GG,5)\}, \{(GT,5)\}, \{(TGG,4), (A,1)\}, \{(C,2), (GC,2), (GA,1)\}, \{(TGC,2), (TGA,1)\}\}$.

3.3 Horizontalna delitev

V drugi fazi poteka gradnja priponskih poddreves T_{p_1}, T_{p_2}, ..., T_{p_n}. Preden algoritem ERA začne graditi drevo, najprej pripravi novo vmesno podatkovno strukturo. S tako delitvijo doseže, da se lokalizirajo dostopi v pomnilnik. S tem se izognejo dragi obhodi po še nezgrajenem poddrevesu, in sicer vsakokrat, ko se vstavlja novo vozlišče. Algoritem ERA razdeli razpoložljiv pomnilnik na fiksni in variabilni del, kot prikazuje Slika 3.2.



Slika 3.2: Razporeditev razpoložljivega pomnilnika.

V fiksnem delu si delijo prostor podatkovna struktura R, ki hrani podnize, trenutni pomnilnik BS velikosti večkratnika bloka datotečnega sistema in manjši del (1 MB), namenjen drevesu predpon, ki povezuje vsa priponska poddrevesa T_{p_1}, T_{p_2}, ..., T_{p_n}. Variabilni del pomnilnika hrani vmesne podatkovne strukture (L, B, I, A, in P), in priponsko poddrevo T_p, v razmerju 40:60.

3.3.1 Pripravljalna faza

V pripravljalni fazi Algoritem 2 pripravi pet seznamov. Seznam I hrani indekse zunanjih vozlišč, kot se nahajajo v prvotnem besedilu. Seznam A hrani informacije o tem, ali v procesu izvajanja obstajata dva ali več enakih in aktivnih oziroma neobdelanih elementov. V seznamu R so shranjeni podnizi pripon posameznih priponskih poddreves. Dolžino podnizov Algoritem 2 izračuna z vsako novo iteracijo, tako da pred-definirano velikost seznama R deli s številom elementov, ki pripadajo

aktivnim poljem seznama A. Od tod tudi ime algoritma *elastic range*. Seznam L hrani vrednosti zunanjih vozlišč, to je indekse, kjer se nahajajo predpone p priponskega poddrevesa T_p. Seznam P hrani in vzdržuje indekse zunanjih vozlišč, ki so bili zapisani v seznamu L pred začetkom izvajanja Algoritma 2. Seznam B hrani informacije, na katerem mestu se razdeli vozlišče. Vrednost $B[i] = (c_1, c_2, offset)$, kjer velja $1 \leq i < |L|$, je predstavljena v obliki trojke podatkov. Prva dva podatka sta črki iz abecede in $offset$, odmik števila znakov od indeksa zunanjih vozlišč $L[i]$ in $L[i-1]$, kjer se nahajata črki. Po končani pripravljalni fazi se v razpoložljivem pomnilniku variabilnega dela hranita samo seznama L in B, ki ju Algoritem 3 potrebuje v fazi gradnje poddrevesa.

Za ponazoritev pripravljalne faze je, iz končnega seznama predpon $P' = \{$(GG,5), (GT,5), (TGG,4), (C,2), (GC,2), (TGC,2), (A,1), (GA,1), (TGA,1)$\}$, izbrana predpona GG, ki se v nizu S pojavlja na petih različnih mestih. Za lažjo predstavo so vse priponske besede z izbrano predpono prikazane v Tabeli 3.2.

i	S_i	Priponska beseda
1	S_1	GGTGGTGGTGCGGTGATGGTGC$
4	S_4	GGTGGTGCGGTGATGGTGC$
7	S_7	GGTGCGGTGATGGTGC$
12	S_{12}	GGTGATGGTGC$
18	S_{18}	GGTGC$

Tabela 3.2: Priponske besede s skupno predpono GG.

Tabela 3.3 prikazuje delovanje Algoritma 2 po korakih, iz katere je razvidno, kako se vrednosti seznamov I, A, R, P, L in B spreminjajo. Za vhodna podatka Algoritma 2 sta izbrana beseda S in predpona $p = GG$.

Začetni korak Algoritma 2 predvideva inicializacijo seznamov L, B, I, A, R, P in določitev vrednosti spremenljivke $start = 2$. V seznamu L se zapišejo indeksi, kjer se nahajajo priponske besede predpone GG. Seznama R in B sta prazna, seznam A ima začetne vrednosti enake 0, ostali pa hranijo vrednosti od $0 \ldots L[i]-1$. Algoritem 2 se izvaja, dokler obstaja še nedefinirana vrednost v seznamu B.

Korak 1					
I	0	1	2	3	4
A	0	0	0	0	0
R	TGGT	TGGT	TGCG	TGAT	TGC$
P	0	1	2	3	4
L	1	4	7	12	18
B	∅				
Korak 2					
I	3	4	1	0	2
A	0	0	0	1	1
R	TGAT	TGCG	TGC$	TGGT	TGGT
P	3	2	4	0	1
L	12	7	18	1	4
B	∅				
Korak 3					
I	3	4	*done*	*done*	*done*
A	*done*	*done*	*done*	1	1
R	TGAT	TGCG	TGC$	TGGT	TGGT
P	3	2	4	0	1
L	12	7	18	1	4
B	∅	(A,C,4)	(G,$,5)	(C,G,4)	
Končne vrednosti					
R	TGAT	TGCG	TGC$	GGTGCGGTGA	GCGGTGATGG
P	3	2	4	1	0
L	12	7	18	4	1
B	∅	(A,C,4)	(G,$,5)	(C,G,4)	(C,G,7)

Tabela 3.3: Pripravljalna faza po korakih.

Glavna zanka je razdeljena na tri večje dele. Najprej Algoritem 2 izračuna vrednost spremenljivke *range*, to je dolžino predpone priponske besede. Vzemimo, da je velikost podatkovne strukture $|R|$ enaka 20, ter da je število aktivnih elementov

AE enak 5. Spremenljivka *range* se izračuna z enačbo:

$$range = \frac{|R|}{AE} = 4$$

Prva notranja zanka preveri vrednosti v seznamu I. V primeru, da vrednost ni bila uporabljena, se v $R[I[i]]$ vpiše predpona pripone dolžine *range*, ki se začne na indeksu $L[I[i]] + start$. Druga notranja zanka preverja aktivne elemente glede na seznam A in poskrbi, da se seznami R, P in L uredijo tako, da so vrednosti v seznamu R urejene po abecedi. Pri tem Algoritem 2 ohranja indekse v I. Po končanem urejanju se nove vrednosti I določijo z enačbo:

$$I[P[i]] = i, 0 \leq i < |L|$$

V primerih, ko se v seznamu R pojavita vsaj 2 enaki predponi, se vrednosti v seznamu A, na indeksih, kjer se nahajajo, določijo nove aktivne vrednosti. Pri preverjanju obstoja enakih predpon v seznamu R, se preverja na aktivnih elementih seznama A. Aktivni elementi so določeni s številskimi vrednostmi, neaktivni pa opisno z besedo *done*.

Tretja notranja zanka preverja, ali obstaja nedefinirana vrednost v seznamu B. V primeru, da obstaja se določi skupna predpona cs podnizov seznama R na indeksih $i - 1$ in i, kjer velja $1 \leq i < |L|$. V primeru, da je $|cs| < range$, Algoritem 2 nastavi vrednost na i-tem v seznamu B. Kjer so izpolnjeni pogoji nastavi vrednosti seznamov I in A, kot neaktivne.

Po končanem tretjem koraku je vrednost v seznamu B, na indeksu 4, ostala nedefinirana, zato se postopek ponovi. Ostaneta še dva aktivna elementa, zato je potrebno ponovno izračunati vrednost spremenljivke *range*. Tokrat je vrednost enaka 10, medtem ko spremenljivka *start* hrani vrednost 6.

3.3.2 Faza gradnje priponskega poddrevesa

V fazi gradnje priponskega poddrevesa se uporabijo vrednosti seznamov L in B, ki jih je Algoritem 2 izračunal v pripravljalni fazi, za predpono GG.

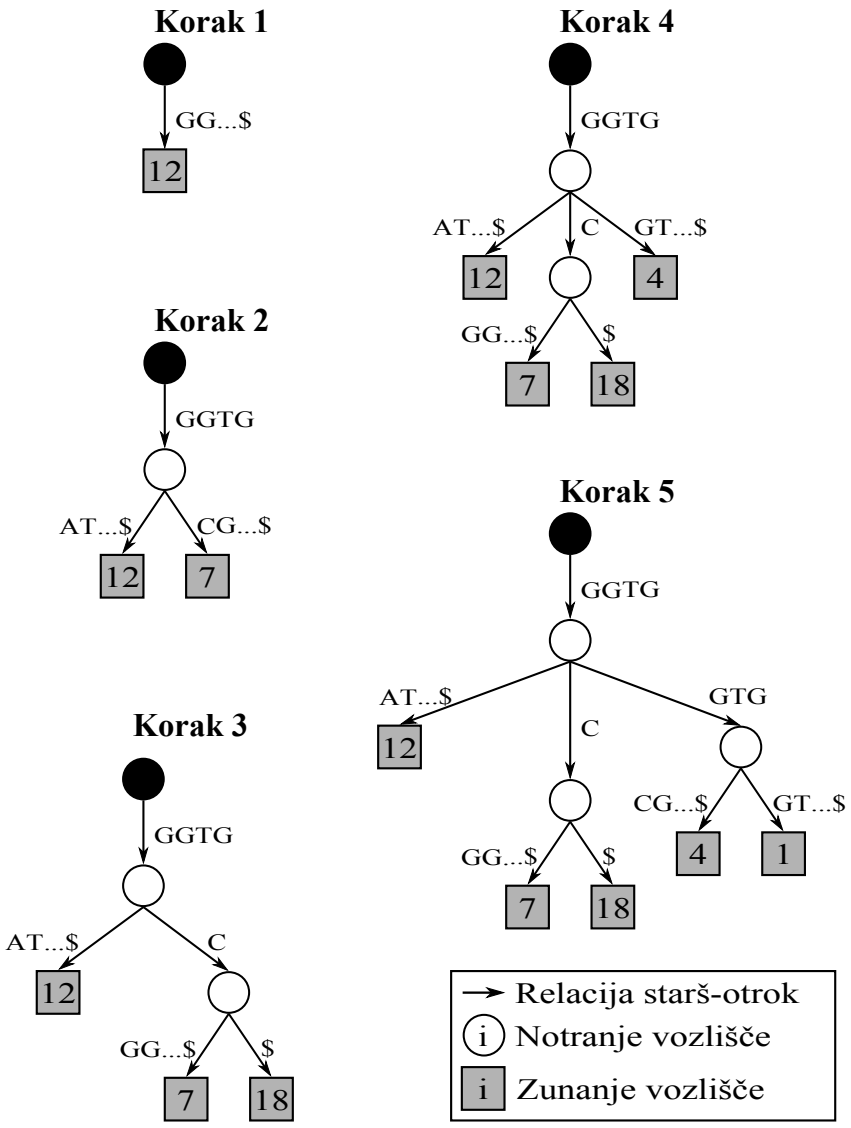
Algoritem 3 za gradnjo priponskega poddrevesa, najprej ustvari povezavo e', ki povezuje koren drevesa *root* s prvim zunanjim vozliščem $u' = L[0] = 12$. Povezavo e' označi s priponsko besedo GGTGATGGTGC\$, ki se nanaša na $L[0]$ in jo vstavi

v sklad. Določi se tudi spremenljivka *depth*, ki hrani vrednost dolžine priponske besede povezave e'. V konkretnem primeru je vrednost enaka 12 (glej Tabelo 3.3).

Zanka Algoritma 3 je razdeljena na tri dele. Prvi del predstavlja ponavljajočo zanko, ki omogoča sprehod po drevesu od zunanjega vozlišča proti korenu drevesa. Ustavi se na tistem vozlišču, ki ne deli več skupne predpone z naslednikom oziroma nima naslednika, trenutno vozlišče pa ne predstavlja zunanjega vozlišča. To določa pogoj drugega dela zanke, ki preverja ali je spremenljivka *depth* enaka spremenljivki $offset$. Če enakost drži, se vozlišču doda povezava do enega naslednika, v nasprotnem primeru se dodata dve novi vozlišči. Prva povezava se označi od indeksa, ki ga hrani $L[i] + depth$ z dolžino $offset - depth$ in doda v sklad. Iz vozlišča na katero je povezana prva povezava sledi druga, ki se označi z ostalim delom pripone od mesta, ki ga določa $offset$.

V tretjem delu Algoritma 3 se na vozlišču, na katero kaže prva povezava, doda nova, z oznako od indeksa, ki ga hrani $L[i] + depth$ z dolžino $offset - depth$ in doda v sklad.

Iz Slike 3.3, ki prikazuje gradnjo priponskega drevesa po korakih, je razvidno, da so pripone besedila, ki se začnejo s predpono GG urejene po abecednem vrstnem redu. Povezave med notranjimi vozlišči hranijo oznake pripon, ki si delijo iste predpone, medtem ko povezave med notranjimi in zunanjimi vozlišči hranijo preostali del pripon. Zunanja vozlišča hranijo indekse, na katerem mestu v besedilu se nahajajo pripone.

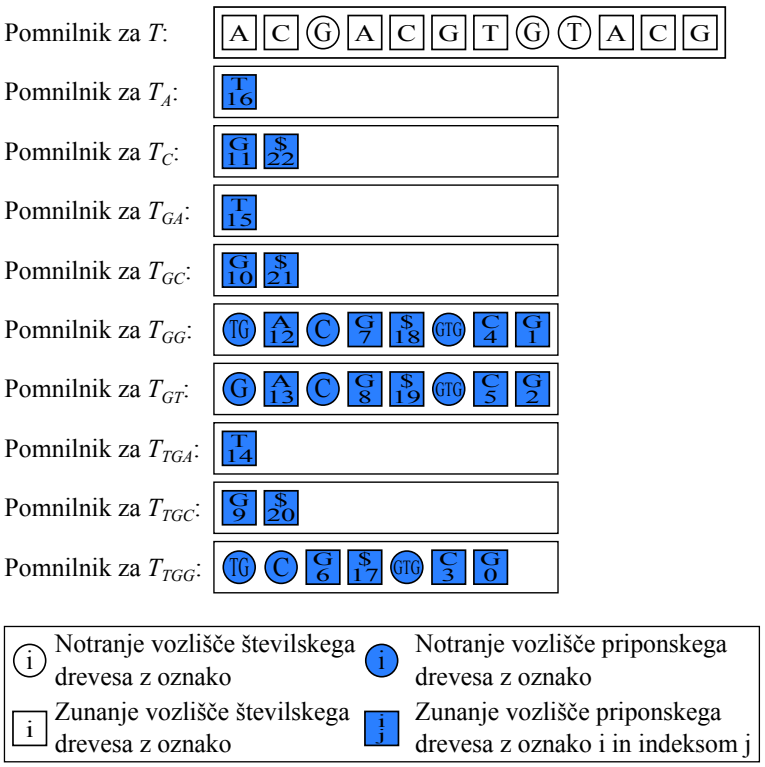


Slika 3.3: Gradnja priponskega drevesa za predpono *GG*.

3.4 Zapis priponskega drevesa na disk

Preden algoritem ERA začne graditi priponsko drevo, shrani izvorno besedilo v novo binarno datoteko. Vertikalna delitev gradnje priponskega drevesa ustvari seznam predpon. Predpone so shranjene v številskem drevesu, njihovo število pa določa število priponskih dreves. Algoritem ERA v datoteko zapiše vrednosti

vozlišč številskega drevesa predpon po principu iskanja v globino. Na enak način stori za posamezna priponska drevesa. Datoteka, v katero je zapisano priponsko drevo, je poimenovana po predponi priponskega drevesa. Slika 3.4 prikazuje primer zapisa vseh predpon in priponskih dreves na disk, ki jih ustvari algoritem ERA, glede na besedilo S, uporabljeno v Podpoglavju 3.2. Zunanje vozlišče priponskega drevesa je prikazano tako, da hrani naslednji znak pripone, ki sledi od starša, ter indeks pripone.



Slika 3.4: Zapis številskega drevesa predpon in priponskih dreves na disk.

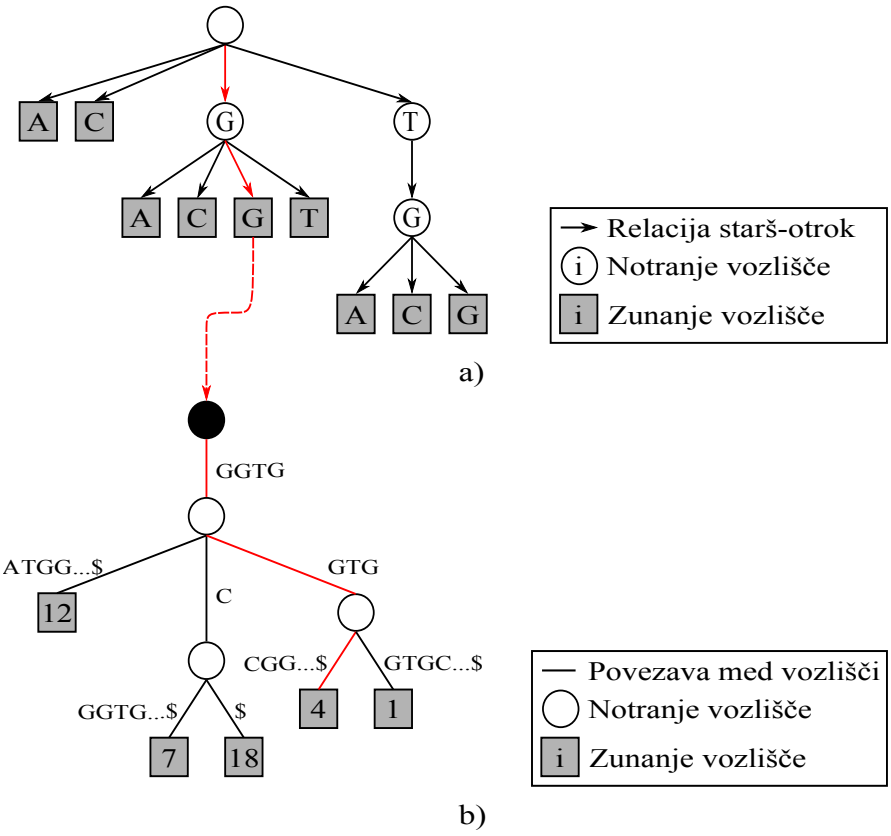
3.5 Iskanje v priponskem drevesu

Za ponazoritev postopka iskanja niza S_f = GGTGGTGCGGTGATGGTGC je uporabljeno priponsko drevo T_{GG}, ki je zgrajeno v Poglavju 3.3.2.

Algoritem ERA najprej z diska v pomnilnik naloži številsko drevo s predpo-

nami. Nato znak za znakom iz S_f preverja, če se nahaja v drevesu predpon, kot prikazuje Slika 3.5 a). Če se tam nahaja nahaja predpona, algoritem naloži priponsko drevo, ki ustreza predponi GG, kot je prikazano na Sliki 3.5 b).

Iskanje po T_{GG} je samo še stvar preverjanja povezav. Funkcija iskanja niza S_f vrne indeks, na katerem se začne v originalnem besedilu S. Iskanje se izvede v $O(log(n) \cdot |S_f|)$ času, kjer je n število naslednikov.



Slika 3.5: Primer iskanja v a) drevesu predpon in b) priponskem drevesu.

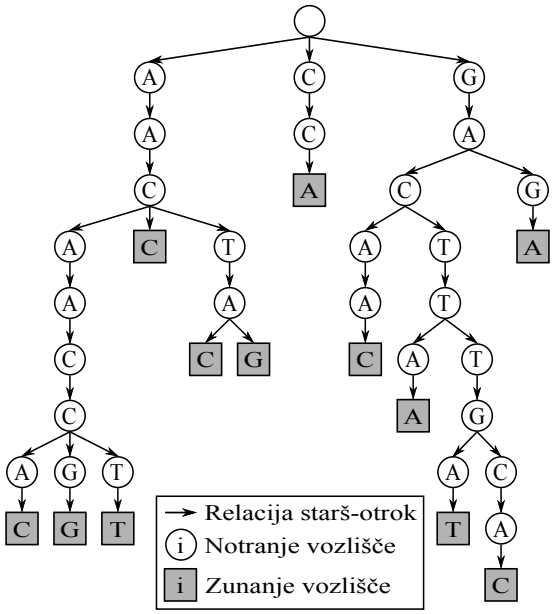
Poglavje 4

COSD

Poglavje povzema delovanje algoritma za gradnjo podatkovne strukture predpomnilniško nezavednega slovarja nizov (ang. *Cache-Oblivious String Dictionary*) [7], shranjeno na disk. Ko je podatkovna struktura COSD (v nadaljevanju struktura COSD) zgrajena, je ni mogoče več spreminjati z brisanjem ali vstavljanjem novih besed. Implementirana je samo operacija iskanja nizov. Algoritem za gradnjo strukture COSD (v nadaljevanju algoritem COSD) se od algoritma ERA razlikuje v tem, da nima informacije o tem, koliko ima na voljo prostega pomnilnika. Gradnja strukture COSD za dolga besedila je zelo dolgotrajen postopek, saj ne izkorišča principa lokalnosti podatkov. Prednost algoritma COSD je v tem, da ga je mogoče implementirati za skoraj vse računalniške sisteme. Uporaba podatkovnih struktur žiraf in slepih dreves ter način zapisa strukture COSD na disk pa omogoča, da operacije iskanja nizov uspešno izkoriščajo princip lokalnosti podatkov in minimizirajo prenose blokov podatkov med različnimi nivoji pomnilnikov (v nadaljevanju V/I dostopi). Slepo drevo dejansko predstavlja številsko drevo s stiskanjem poti. Podatkovna struktura žirafa je podrobneje predstavljena v Poglavju 4.3.

Algoritem COSD najprej zgradi številsko drevo T. Nato ga razčleni na več manjših poddreves (komponente), ki so prek podatkovne strukture most povezana z uravnoteženim iskalnim drevesom. Vsaka komponenta je razdeljena na vsaj eno plast. Posamezna plast komponente vsebuje žirafo in slepo drevo. Slika 4.1 prikazuje številsko drevo T, ki služi kot primer za prikaz gradnje strukture COSD.

Struktura vozlišča v številskega drevesa T hrani spremenljivke, kot so: seznama notranjih in zunanjih vozlišč, reference na mostove, žirafe in slepa drevesa, številka



Slika 4.1: Vhodni podatek algoritma COSD.

komponente in plasti, rang in globino vozlišča, število zunanjih vozlišč glede na celotno drevo in glede na posamezno komponento, ter vrednost, ali gre za zunanje vozlišče in oznako vozlišča.

Podanih je nekaj osnovnih oznak in njihov pomen, ki se uporabljajo v nadaljevanju:

- v označuje vozlišče drevesne strukture T
- T_v označuje poddrevo drevesa T s korenom v vozlišču v,
- n_v označuje število zunanjih vozlišč poddrevesa T_v,
- $depth(v)$ označuje globino vozlišča v oziroma število povezav na poti od vozlišča v do korena drevesa T,
- $rank(v)$ označuje rang vozlišča v, ki se izračuna z naslednjo enačbo:

$$rank(v) = \begin{cases} 0 & \text{if } n_v = 0 \\ \lceil log(n_v) \rceil & \text{else} \end{cases}$$

Gradnja strukture COSD poteka v štirih fazah, ki so opisana po posameznih podpoglavjih. V prvi fazi algoritem COSD zgradi številsko drevo T in v vozlišča zapiše vrednosti n_v, $depth(v)$ in $rank(v)$. Nadaljuje z razčlenitvijo T v komponente. Druga faza poskrbi za gradnjo drevesa komponent. V tretji fazi se za vsako

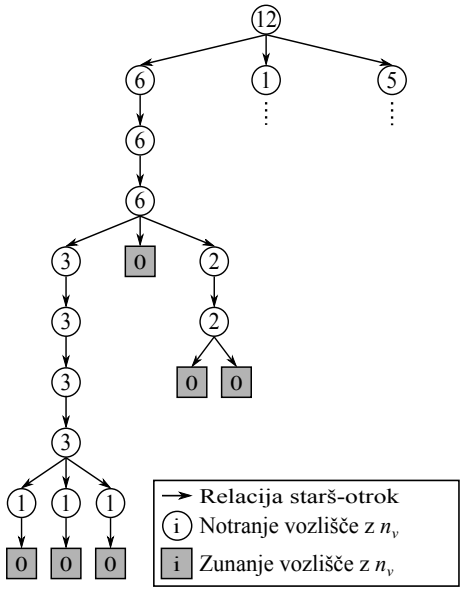
plast v komponenti zgradijo žirafe in slepa drevesa. Sledi opis faze zapisa strukture COSD na disk. V zadnjem podpoglavju je predstavljen primer iskanja niza v strukturi COSD.

4.1 Gradnja številskega drevesa

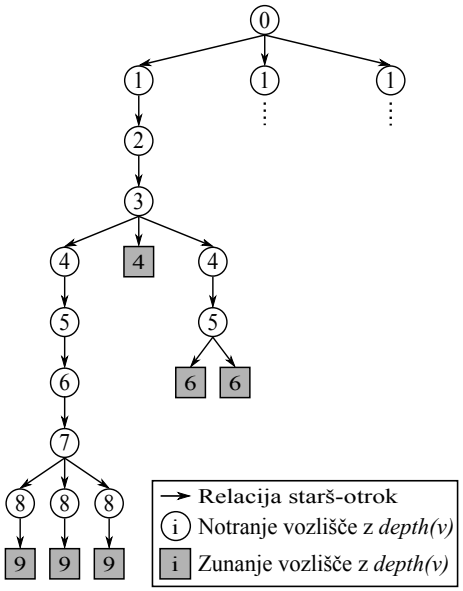
Algoritem COSD za vhodni podatek prejme na disku shranjeno datoteko, v kateri so zapisani nizi, in sicer vsak niz v svoji vrstici. Ko je številsko drevo T zgrajeno, se algoritem COSD večkrat zapored sprehodi po vseh vozliščih z namenom, da zapiše v vozlišča vrednosti števila zunanjih vozlišč, globino, rang, razčleni drevo T na komponente, ter jih uredi.

4.1.1 Zapis vrednosti v vozlišča drevesa T

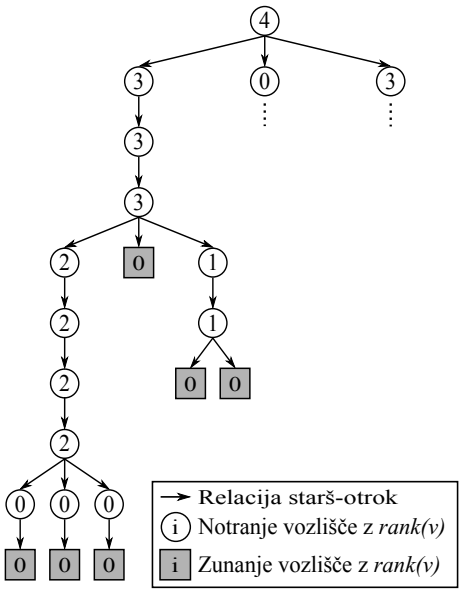
Algoritem COSD se v prvem koraku, sprehodi čez vsa vozlišča, v katera zapiše vrednosti števila zunanjih vozlišč, globine in ranga. Slika 4.2 prikazuje primer števila zunanjih vozlišč, Slika 4.3 globino in Slika 4.4 rang za vozlišča skrajno levega poddrevesa drevesa T.



Slika 4.2: Vozlišča z n_v skrajno levega poddrevesa.



Slika 4.3: Vozlišča z $depth(v)$ skrajno levega poddrevesa.



Slika 4.4: Vozlišča z $rank(v)$ skrajno levega poddrevesa.

4.1.2 Razčlenitev na komponente

Algoritem COSD se z namenom razčlenitve drevesa T na komponente sprehodi čez vsa vozlišča. Tako se posameznemu vozlišču določi vrednost komponente in plasti, kateri pripada. Na začetku r predstavlja koren drevesa T in koren prve komponente. V nadaljevanju r predstavlja samo koren komponent. Vozlišče v pripada plasti 0, če je razlika globine vozlišča v in globina vozlišča r manjša od 2. V nasprotnem primeru vozlišče v pripada plasti i, če je izpolnjen naslednji pogoj

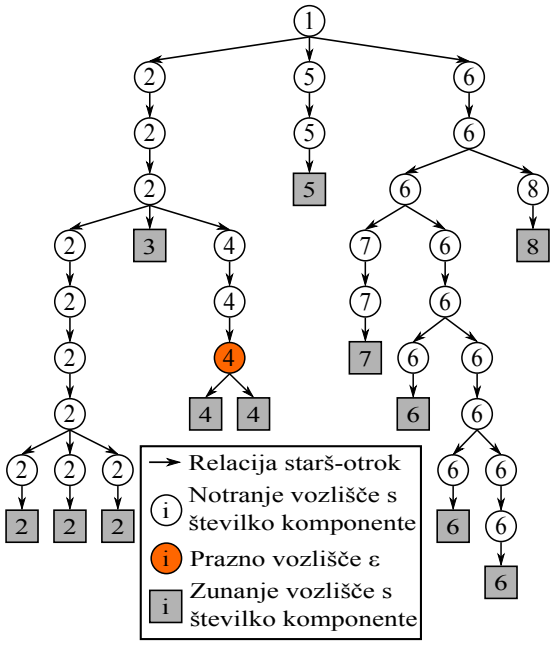
$$2^{2^{i-1}} \leq depth(v) - depth(r) < 2^{2^i}$$

za $i = 1, 2, \ldots$. S pogojem

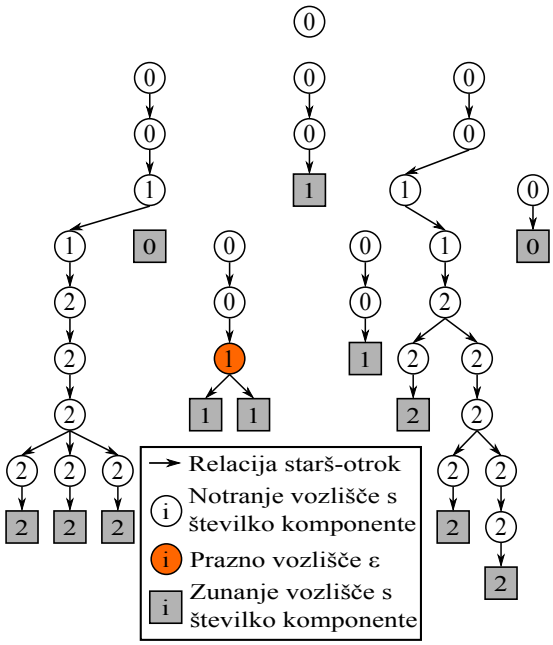
$$rank(r) - rank(v) < \varepsilon 2^i$$

se preveri, ali je trenutno vozlišče v kandidat za komponento. V primeru, da pogoj ne drži, se ustvari nova komponenta s korenom r v trenutnem vozlišču in novo vrednostjo. Drugače se v vozlišče shrani zaporedna vrednost komponente ter vrednost plasti i. Iz Slike 4.5 je razvidno, da je največja vrednost komeponente enaka 8, kar pomeni, da je drevo T razčlenjeno na 8 komponent.

Z izbiro vrednosti konstante $\varepsilon \in v(0, 1]$ se vpliva na velikost komponente. Nižja je vrednost konstante, manj je vozlišč v komponenti, in obratno. V našem primeru je bila izbrana vrednost $\varepsilon = 1$. Slika 4.6 prikazuje komponente drevesa T s posameznimi vrednostmi plastmi.



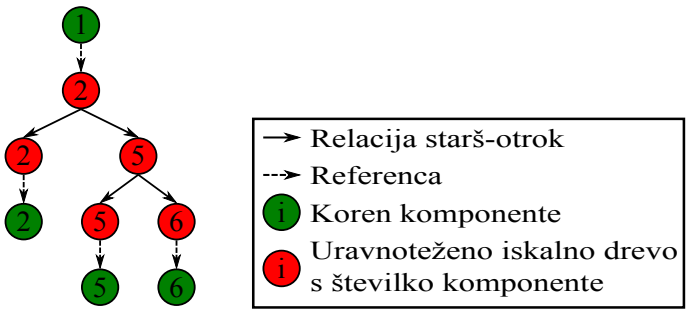
Slika 4.5: Vozlišča drevesa T z vrednostmi komponent.



Slika 4.6: Vozlišča drevesa T z vrednostmi plasti.

4.1.3 Urejanje komponent

Ko so vse komponente identificirane, sledi tretji korak, v katerem se algoritem ponovno sprehodi po drevesu T. Pri tem za vsako posamezno vozlišče zapolnita in po velikosti uredita seznama notranjih in zunanjih vozlišč. V seznamu zunanjih vozlišč so shranjeni koreni komponent, v seznamu notranjih vozlišč pa vozlišča znotraj komponente. Vozlišča, ki predstavljajo korene komponent, je potrebno povezati med seboj. Zunanja vozlišča se shranijo v podatkovni strukturi most, ki jih poveže po principu uravnoteženega iskalnega drevesa ali optimalnega binarnega drevesa. Slika 4.7 prikazuje primer zunanjih vozlišč, povezanih po principu uravnoteženega iskalnega drevesa.



Slika 4.7: Uravnoteženo iskalno drevo za zunanja vozlišča.

Posamezna komponenta je razdeljena na plasti, znotraj katere lahko obstaja vozlišče z več kot enim naslednikom v naslednji plasti. V tem primeru se vstavi prazno vozlišče ε med vozliščem in nasledniki. Vozlišče mora pripadati isti plasti kot nasledniki. Primer praznega vozlišča ε je prikazan na Sliki 4.5.

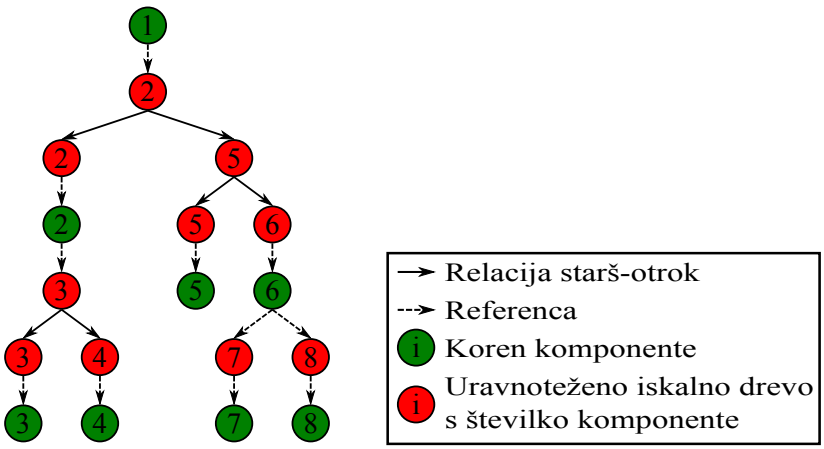
V tretjem koraku algoritem COSD poskrbi, da v vseh vozliščih, ki predstavljajo koren določene plasti, ustvari slepo drevo. Naslednikom iste plasti pa poda referenco na slepo drevo starša.

V zadnjem koraku se posameznim vozliščem določi število zunanjih vozlišč na trenutni plasti.

4.2 Gradnja drevesa komponent

Struktura COSD je zapisana na disk po *vanEmdeBoas* razporeditvi, ki za vhodni podatek potrebuje dvojiško iskalno drevo. Zato je potrebno komponente drevesa T, predstaviti v obliki dvojiškega iskalnega drevesa komponent T'. Razporeditev strukture COSD na disk po *vanEmdeBoas* je bolj podrobno predstavljena v Poglavju 4.4.

Ko algoritem COSD gradi drevo T', se za vsako komponento identificirajo tista vozlišča znotraj komponente, ki imajo naslednike v drugi komponenti. Ta vozlišča se uporabijo za gradnjo dvojiškega uravnoteženega iskalnega drevesa, ki predstavljajo posamezno komponento. Utež vozlišča predstavlja vsota končnih vozlišč naslednikov v drugi komponenti. Tem vozliščem so podane reference na podatkovno strukturo most. Za gradnjo dvojiškega iskalnega drevesa se lahko uporabi algoritem po principu uravnoteženega iskalnega drevesa ali Huffmanov algoritem. Slika 4.8 prikazuje končno obliko drevesa komponent T' za podani primer na Sliki 4.1.



Slika 4.8: Drevo komponent T'.

4.3 Gradnja žiraf in slepih dreves

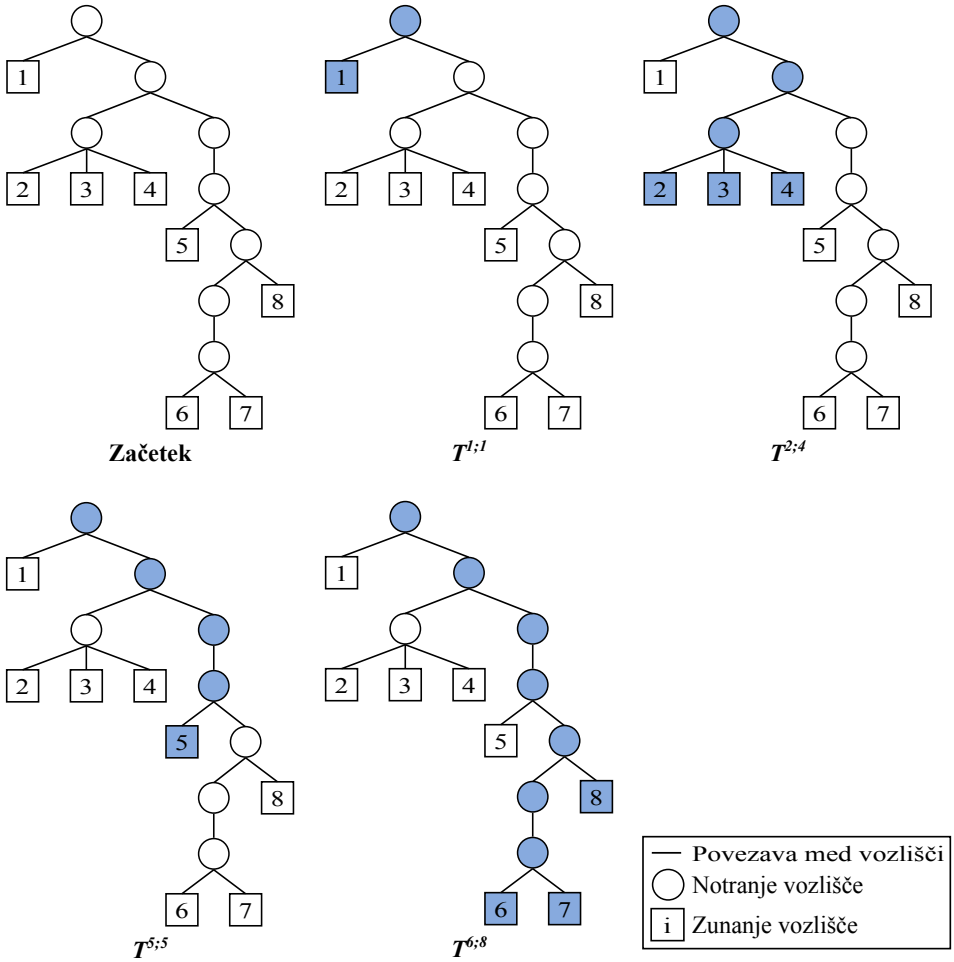
Posamezna komponenta vsebuje najmanj eno plast, vsaka plast pa vsebuje dve podatkovni strukturi: žirafo in slepo drevo. Slepo drevo kaže na žirafo, žirafa pa na naslednje slepo drevo. Žirafa zadnje plasti v komponenti nima neposredne reference na slepo drevo prve plasti druge komponente. Zato je uvedena podatkovna struktura most, ki to omogoča.

4.3.1 Žirafa - drevesna podatkovna struktura

Žirafa kot drevesna podatkovna struktura je definirana kot drevo, katerih zunanja vozlišča si delijo vsaj polovico notranjih vozlišč. Drevo T je lahko razstavljeno na več žiraf na različne načine. Algoritem 4 gradi žirafe s požrešno metodo.

Naj $T^{i;j}$ predstavlja drevo z zunanjimi vozlišči od i do j, kjer $1 \leq i \leq n$, $j < n$ in n je število vseh zunanjih vozlišč drevesa T. Algoritem 4 bere zunanja vozlišča od leve proti desni in pri tem vzdržuje seznam zunanjih vozlišč L, ki pripadajo posamezni žirafi. V vsaki iteraciji preveri, ali naslednje zunanje vozlišče pripada trenutni žirafi. V primeru, da pripada, se zunanje vozlišče doda v seznam L. Če ne, vrne žirafo, odstrani vse elemente iz seznama L in vanj doda trenutno zunanje vozlišče.

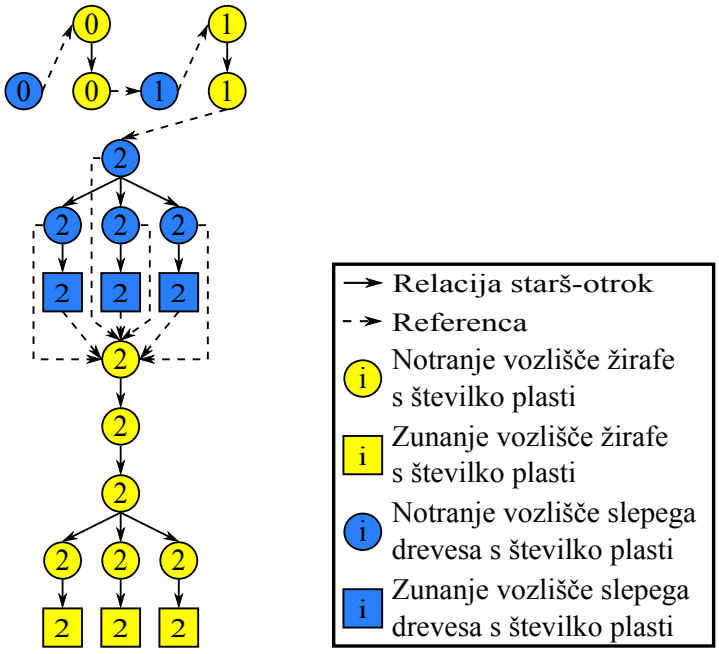
Vsako zunanje vozlišče drevesa T je vsebovano v žirafi le enkrat, medtem ko se notranja vozlišča lahko pojavijo večkrat. Iz Slike 4.9, ki prikazuje primer gradnje žiraf s požrešno metodo, je razvidno, da je Algoritem 4 iz drevesa T zgradil pet žiraf, katerih vozlišča so obarvana z modro bravo.



Slika 4.9: Gradnja žiraf s požrešno metodo.

4.3.2 Slepo drevo - številsko drevo s stiskanjem poti

Slepo drevo dejansko predstavlja številsko drevo s stiskanjem poti, vendar je za gradnjo strukture COSD prilagojena, saj dodatno vsebuje reference na žirafo. Slepo drevo v svoji strukturi hrani seznam svojih naslednikov, referenco na žirafo, oznako in število opuščenih znakov. Algoritem COSD gradi slepa drevesa za vsako plast v komponenti. Slika 4.10 prikazuje primer povezanosti žiraf in slepih dreves znotraj komponente 2. Kot je razvidno iz Slike 4.10 ima komponenta tri plasti in vsaka plast vsebuje slepo drevo in žirafo.

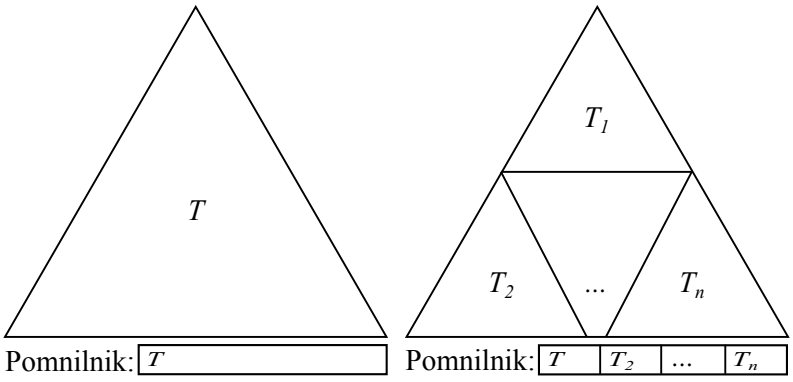


Slika 4.10: Povezanost žiraf in slepih dreves znotraj komponente.

4.4 Zapis podatkovne strukture COSD na disk

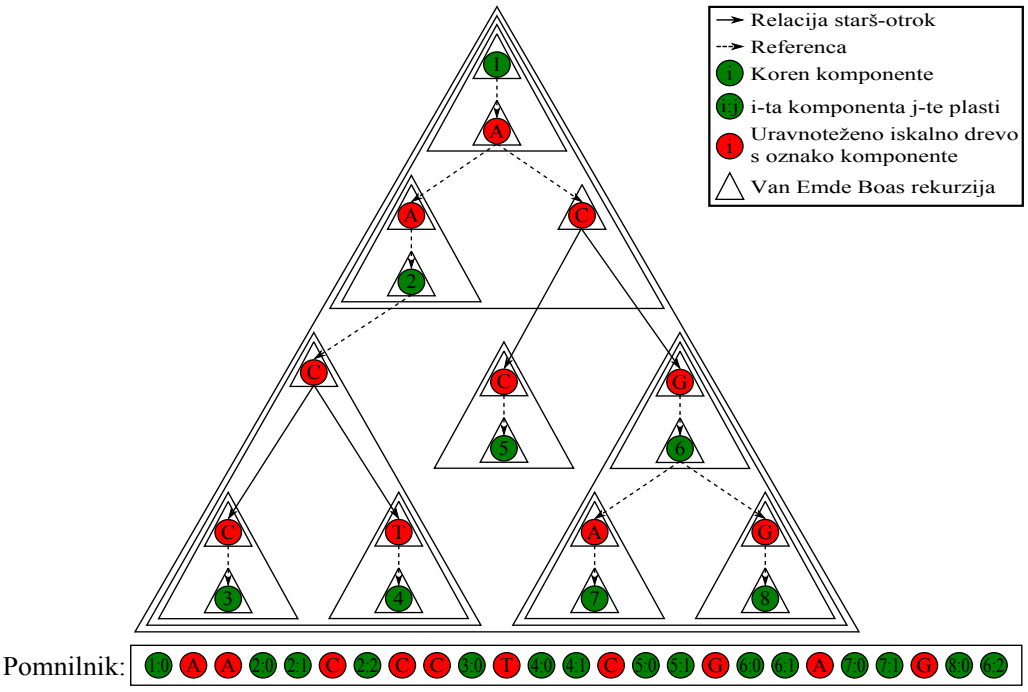
Za zapis strukture COSD na disk je uporabljen algoritem za zapis po *van Emde Boas*, katerega koncept delovanja je prikazan na Sliki 4.11. Gre za rekurzivno metodo, ki celotno drevo T razpolovi na zgornje drevo T_1 in spodnja drevesa $T_2, \ldots, T_n$. Nato rekurzivno nadaljuje z zgornjim drevesom, sledijo mu spodnja drevesa, od leve proti desni. Postopek se konča, ko višina drevesa doseže določen pogoj, na primer, ko je višina drevesa enaka ena. Tako kot se metoda rekurzivno sprehaja po drevesu, se na enak način vrednosti žiraf, slepih dreves in mostov, zapisujejo na disk.

Pri uporabi metode zapisa drevesa komponent na disk po *van Emde Boas*, se z globino rekurzivnega klica določi vrstni red zapisa različnih plasti posamezne komponente. Globina rekurzivnega klica je oštevilčena v obratnem vrstnem redu, z začetkom najbolj notranjega rekurzivnega klica. Razlog takšnega določanja globine rekurzivnih klicev je v tem, da se globina rekurzivnega klica ujema s plastjo posamezne komponente, ko se te zapisujejo na disk. Na primer, ko se v rekurziji



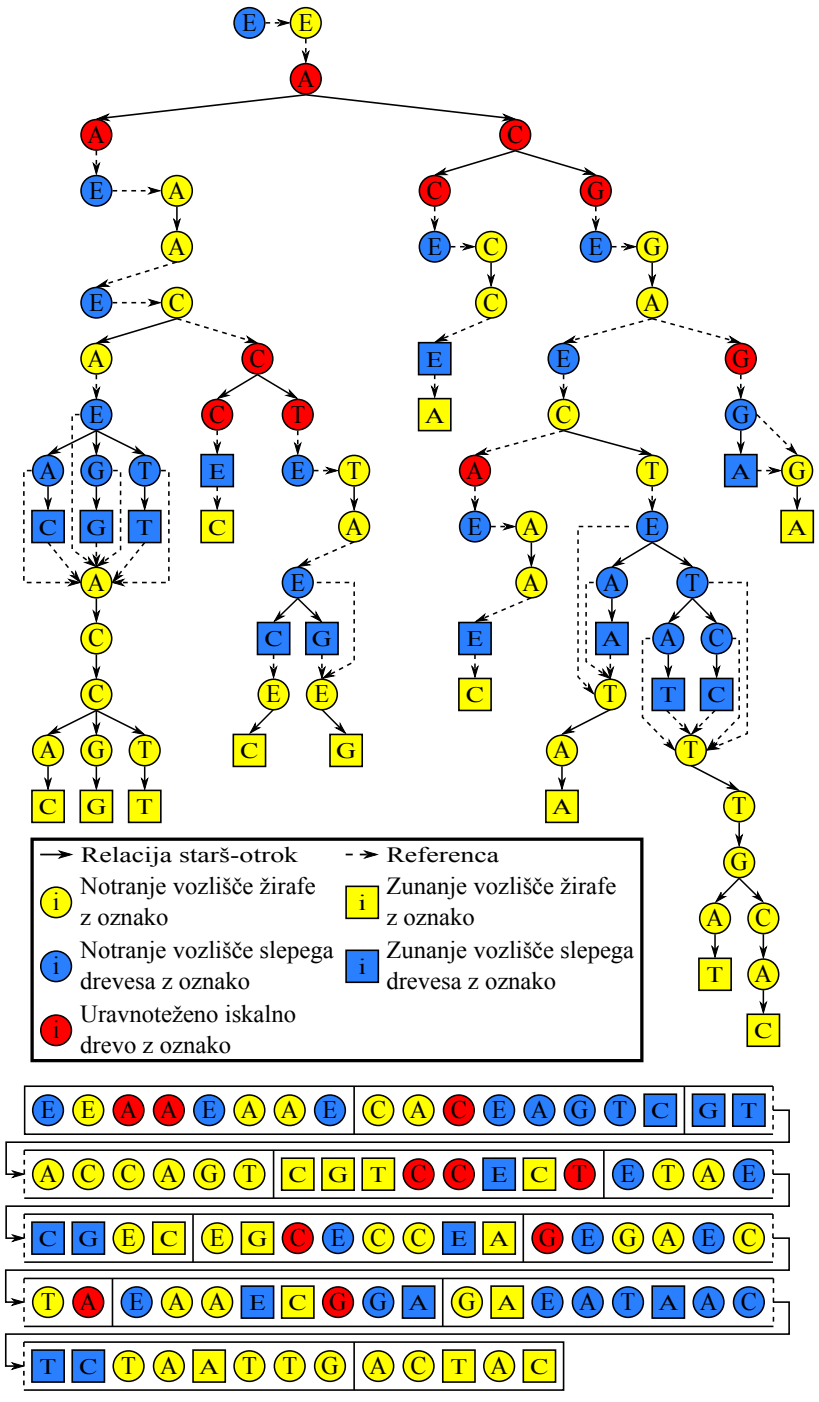
Slika 4.11: Zapis splošne podatkovne strukture na disk po *vanEmdeBoas*.

pojavi koren komponente na globini 0, se plast 0 zapiše na disk. Na globini 1 se zapiše plast 1, in tako naprej. V primeru, da plast i na globini i ne obstaja, se ta izpusti. Slika 4.12 prikazuje primer metode van Emde Boas za drevo komponent.



Slika 4.12: Zapis drevesa komponent T' na disk po *vanEmdeBoas*.

Vsaka plast je sestavljena iz slepih dreves in žiraf, ki se zapisujejo na disk po metodi iskanja v širino. Najprej se zapiše slepo drevo, nato žirafe. Celotno strukturo COSD in način zapisa na disk, prikazuje Slika 4.13. Črka E označuje prazno vozlišče. Podatki strukture COSD se dejansko zapišejo v datoteko shranjeno na disk. V prvo vrstico se zapiše velikost celotne strukture COSD v bajtih. Nato se v vsako vrstico zapišejo podatki posamezne strukture, kot si sledijo na spodnjem delu Slike 4.13. Prvi podatek v vrstici se nanaša na identifikacijsko številko podatkovne strukture. Za slepo drevo je to številka 1, za žirafo številka 2 in za uravnoteženo iskalno drevo številka 3. Ko je zapisana identifikacijska številka podatkovne strukture, se zapišejo še ostali podatki, ki jih le-ta hrani. Način zapisa podatkov v datoteko je pomemben za izvajanje poizvedb, kar bo bolj podrobno obrazloženo v naslednjem podpoglavju.



Slika 4.13: Zapis podatkovne strukture COSD na disk po *vanEmdeBoas*.

4.5 Iskanje v podatkovni strukturi COSD

Preden algoritem COSD začne s poizvedbami, naloži datoteko, v kateri so shranjeni podatki strukture COSD, z diska v pomnilnik. Najprej prebere prvo vrstico, ki hrani velikost strukture COSD in rezervira tabelo t te velikosti. Tabela t je tipa *char*. Vse ostale vrstice se nanašajo na podatkovne strukture: slepo drevo, žirafo in uravnoteženo iskalno drevo. Za katero podatkovno strukturo gre, pove prvi podatek v vrstici. Ostali podatki vrstice se zapišejo v t.

Iskanje poteka v treh podatkovnih strukturah: slepem drevesu, žirafi in uravnoteženem iskalnem drevesu. Prvi znak iskanega niza se najprej preveri v slepem drevesu, ki se nahaja na prvem mestu tabele t. Sledi potrditev v žirafi, ki iskanje naslednjega znaka usmeri, neposredno ali preko uravnoteženega iskalnega drevesa, v naslednje slepo drevo.

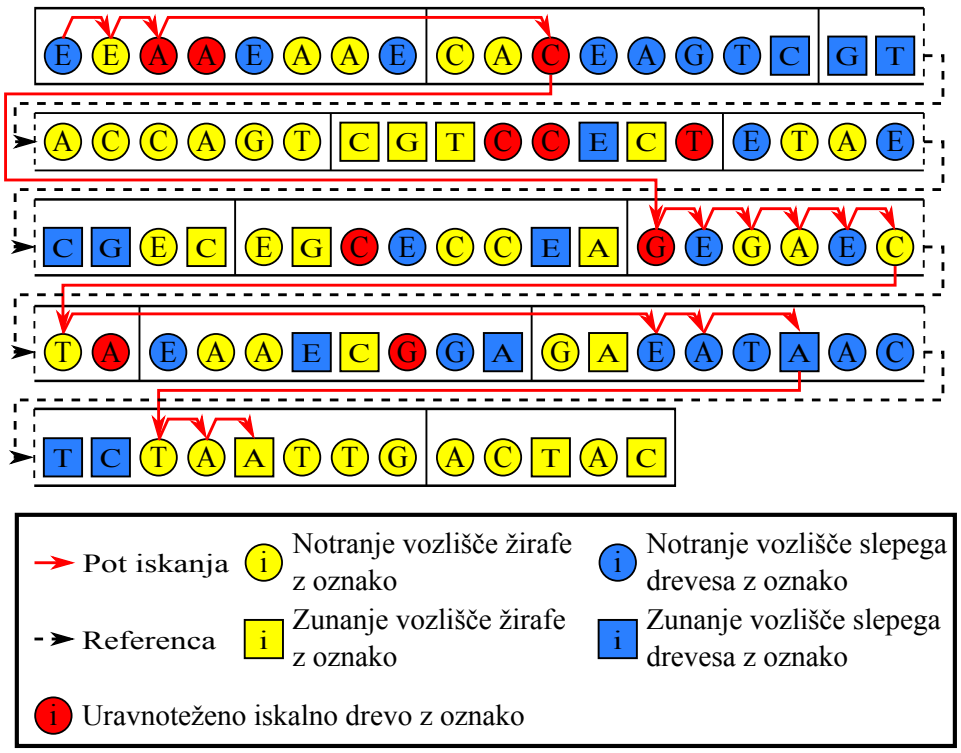
Iskanje v slepem drevesu poteka od korena do zunanjega vozlišča. Koren slepega drevesa je vedno prazno vozlišče E, ki hrani samo število opuščenih znakov. V trenutnem vozlišču se ne preverja ali je znak, ki ga hrani slepo drevo enak znaku v iskanem nizu. To se preverja v naslednikih. Če nobeden od naslednikov ne ustreza, se preišče žirafa, ki jo hrani trenutno vozlišče, v nasprotnem primeru nadaljuje z naslednikom, ki ustreza. Ko je z iskanjem v slepem drevesu doseženo zunanje vozlišče, se preišče žirafa, ki jo hrani. Žirafa hrani vse opuščene znake slepega drevesa od korena do zunanjega vozlišča. Rezultat iskanja v žirafi je lahko koren slepega drevesa ali nič.

Iskanje v žirafi poteka od korena do zunanjega vozlišča. Da bi iskanje doseglo zunanje vozlišče, morata znaka v trenutnem vozlišču in nasledniku ustrezati znakoma v iskanem nizu. Če to drži samo za trenutno vozlišče, se iskanje nadaljuje v uravnoteženem iskalnem drevesu, katerega rezultat je lahko koren slepega drevesa ali nič. V slednjem primeru se iskanje v žirafi zaključi, saj ni bilo mogoče najti iskanega niza. Enak postopek velja, če je doseženo zunanje vozlišče žirafe.

Uravnoteženo iskalno drevo hrani znak, ki se preveri s trenutnim iskanim znakom niza. Iskanje se nadaljuje v levega ali desnega naslednika. Ko je doseženo zunanje vozlišče se ponovno preveri iskani znak. Če se znaka ujemata je rezultat iskanja koren slepega drevesa.

Za ponazoritev primera iskanja niza S_f = GACTTAA je uporabljen primer strukture COSD, ki je predstavljena v Poglavju 4.4. Slika 4.14 prikazuje primer

iskanja niza S_f. Pot iskanja je označena z rdečimi puščicami.



Slika 4.14: Iskanje niza $GACTTAA$ v podatkovni strukturi COSD.

Poglavje 5

Primerjava algoritmov ERA in COSD

Priponsko drevo in struktura COSD sta statični podatkovni strukturi. Omogočene so samo poizvedbe nizov, zato je vsebina poglavja namenjena predvsem časovnim primerjavam poizvedb ter meritvam V/I dostopov v predpomnilnikih.

V poglavje smo vključili tudi časovne in V/I meritve poizvedb podatkovne strukture B–drevo nizov [2], ki je v diplomski nalogi [8] implementirana v dveh verzijah. Obe verziji delujeta v glavnem pomnilniku in se razlikujeta po tem, da prva shranjuje vstavljene nize znotraj strukture, medtem ko druga hrani samo kazalce na vstavljene nize.

5.1 Okolje in podatki

Tehnične podrobnosti strojne opreme računalnika, na katerem je potekala gradnja priponskega drevesa in strukture COSD, in na katerem so se izvajale poizvedbe, prikazuje Tabela 5.1.

	Opis
Model	2 x AMD Opteron 6272 @2.10 GHz (32 jeder)
Arhitektura	x86_64
Predpomnilnik L1d	16 KB
Predpomnilnik L1i	64 KB
Predpomnilnik L2	2048 KB
Predpomnilnik L3	6144 KB
RAM	2 x 8GB DIMM DDR3 Synchronous 1333 MHz (na jedro)
Trdi disk	Seagate Baracuda ST3250310NS, 7200 RPM, 250 GB
OS	Ubuntu 13.10 (GNU/Linux 3.11.0-15-generic x86_64)

Tabela 5.1: Tehnične podrobnosti strojne opreme računalnika.

5.1.1 Programska koda

Izvorna koda algoritma ERA je shranjena v datoteko *era.c* in je dostopna na [12, 14]. Za gradnjo priponskega drevesa smo uporabili naslednjo obliko ukaza:

```
./era <Datoteka s podanimi parametri>
```

Parametrom smo nastavili naslednje vrednosti:

data_file=./<Datoteka z vhodnim nizom>
alpha=ACGT$
working_folder=./dna_out
seek=0
size=0
memory_budget=512
max_tree_size=0
query=GGTG

Za izvajanje večjega števila poizvedb se je algoritem ERA izkazal za neprimernega, zato smo programsko kodo prilagodili, tako da se je struktura naložila samo enkrat in da smo lahko opravili več poizvedb. Poleg tega smo odstranili tudi del originalne izvorne kode, ki se nanaša na gradnjo priponskega drevesa. Programsko

kodo algoritma smo shranili v datoteko *eraSearch.c* in jo uporabili za poizvedbe. Za poizvedbe smo uporabili naslednjo obliko ukaza:

```
./eraSearch <Datoteka z iskanimi nizi>
```

Programa smo prevajali s prevajalnikom gcc (verzija 4.8.2) s stikalom `O3`. Datoteke *readme.txt*, ki opisuje pomen parametrov, *par_dna.txt* in *eraSearch.c* so dostopne na [14].

Izvorne kode algoritma COSD [14] nismo spreminjali. Algoritem se prevede z ukazom `make`, ker sta avtorja [7] pripravila datoteko *Makefile*. Uporabila sta stikala: `Wall`, `O3` in `g` ter prevajalnik g++. Prevajalnik pripravi tri izvršljive datoteke: `trierunner`, `colayouter` in `corunner`.

Za gradnjo strukture COSD smo uporabili ukaz:

```
./colayouter -i <Datoteka z vhodnimi nizi> -o ./<Ime strukture>.veb
-e 1.0 -b 0 -c 0 -v 1 -w 0 -d 0.5
```

Za poizvedbe v strukturi COSD smo uporabili ukaz:

```
./corunner -i <Ime strukture>.veb -t <Datoteka z iskanimi nizi>
-v 0 -r 1 -c 1
```

Za časovne meritve so uporabljene vgrajene funkcije standardnih knjižnic za programski jezik C in C++. Za testiranje obeh algoritmov velja naslednje zaporedje ukazov:

1. naloži strukturo v glavni pomnilnik
2. začni meriti čas
3. izvedi iskanje
4. ustavi merjenje časa

Za meritve V/I dostopov v predpomnilnikih, je uporabljeno orodje *Cachegrind* programskega paketa Valgrind [11]. Orodje *Cachegrind* simulira posamezne nivoje predpomnilniške hierarhije in beleži zgrešene V/I dostope (m) na prvem (L1) in zadnjem (LL) nivoju. V našem primeru obstaja tri-nivojska predpomnilniška hierarhija, kot je prikazana na Sliki 2.4, zato v LL beleži V/I dostope na nivoju L2 in L3. V/I dostopi so lahko bralni (r) ali pisalni (w). Dostopa se lahko do podatkov

(D) ali ukazov programu (I). Medtem ko so dostopi do podatkov lahko bralni (Dr) ali pisalni (Dw), so dostopi do ukazov programu samo bralni (Ir).

Za meritve V/I dostopov poizvedb v strukturi COSD smo uporabili ukaz:

```
valgrind --tool=cachegrind ./corunner -i <Ime strukture>.veb -t
<Datoteka z iskanimi nizi> -v 0 -r 1 -c 1
```

Za meritve V/I dostopov poizvedb v priponskem drevesu smo uporabili ukaz:

```
valgrind --tool=cachegrind ./eraSearch <Datoteka z iskanimi nizi>
```

Program Valgrind zapiše vrednosti V/I dostopov v datoteko *cachegrind.out.PID*, kjer *PID* predstavlja številko procesa. V datoteki so zapisani V/I dostopi posameznih klicev vgrajenih funkcij standardnih knjižnic in funkcij algoritma, tako kot si sledijo ob izvajanju programa. Vrednosti V/I dostopov smo združili skupaj, po posamezni funkciji, z ukazom:

```
cg_annotate --threshold=0 cachegrind.out.PID > <Ime datoteke>
```

Pri predstavitvi rezultatov meritev V/I dostopov poizvedb nismo upoštevali klicev vgrajenih funkcij standardnih knjižnic in tistih funkcij programa, ki se nanašajo na nalaganje strukture v glavni pomnilnik. Tako smo pri poizvedbah v priponskem drevesu upoštevali funkciji `Match` in `ReadBlock`, pri poizvedbah v strukturi COSD pa `isContained` in `searchGiraffe`.

5.1.2 Testni podatki

Za gradnjo podatkovnih struktur smo pripravili vhodna podatka S_1 in S_2, in za testiranje poizvedb štiri testne primere: Q_1, Q_2, Q_3 in Q_4, ki vsebujejo 5.000, 10.000, 15.000 in 20.000 nizov [14]. Osnovo za pripravo vhodnih podatkov predstavlja človeški genom G [12]. Kot vhodni podatek S_1 smo izbrali $G[10^7 \ldots 10^7 + 12.000)$. Vhodni podatek S_2 hrani vse pripone niza S_1.

Za vsak testni primer Q_i velja:

- polovica nizov v Q_i se nahaja v S_1, ostala polovica ne,
- nize v Q_i smo razporedili naključno in
- dolžina posameznega niza v Q_i je v razponu $[4, |S_1|)$ in je naključna z enakomerno porazdelitvijo.

Generiranje testnih primerov je potekalo v dveh fazah. V prvi fazi smo ustvarili tabelo t velikosti števila nizov v Q_i. Vrednosti v t smo nastavili na $false$. Nato smo v zanki generirali naključna števila r, kjer $0 \leq r < |t|$. Če je bila podana enakost $t[r] == false$, smo povečali števec in vrednost $t[r]$ nastavili na $true$. Zanka se je ponavljala toliko časa, dokler je bil števec manjši od polovice števila vseh nizov v Q_i.

V drugi fazi smo se v zanki sprehodili čez vse vrednosti tabele t. Najprej smo generirali naključno število r v razponu $[4, |S_1|)$. Nato smo preverjali vrednost tabele t na trenutnem indeksu j. Če je bila podana enakost $t[j] == true$, smo v Q_i zapisali niz $S_1[r \ldots |S_1|)$. V nasprotnem pa niz, ki ne obstaja v S_1.

Vhodni podatek S_1 je namenjen gradnji priponskega drevesa in obema verzijama B–dreves nizov, S_2 pa strukturi COSD. Algoritem ERA zgradi eno priponsko drevo, velikosti 320 KB, v 0.14 sekundah. Nadalje, gradnja strukture COSD traja 8.3 minute in zasede 2.880 GB pomnilnika. Največji del strukture COSD predstavljajo žirafe, s kar 2.877 GB, sledijo slepa drevesa z 2.57 MB in uravnotežena iskalna drevesa z 0.8 MB (kot primer glej Sliko 4.13). Končno, prva verzija B–drevesa zasede 786.9 MB pomnilniškega prostora in je zgrajena v 0.53 sekundah, medtem ko druga zasede 56.6 MB in je zgrajena v 0.42 sekundah.

5.2 Časovne meritve poizvedb

Velikosti podatkovnih struktur, ki jih bomo primerjali so takšne, da se jih lahko naloži v glavni pomnilnik. Najhitreje se naloži priponsko drevo (0.007 s), ki mu sledi druga verzija B–drevesa nizov (0.61 s). Na tretjem mestu je prva verzija B–drevesa nizov (0.83 s). Struktura COSD se naloži v glavni pomnilnik v 50 sekundah, kar je daleč največ časa v primerjavi z ostalimi strukturami.

Iz Tabele 5.2, ki prikazuje časovne meritev poizvedb po priponskem drevesu, strukturi COSD in obeh verzijah B–drevesa nizov, je razvidno, da so poizvedbe najhitrejše v prvi verziji B–drevesa nizov in najpočasnejše po strukturi COSD. Rezultati meritev poizvedb po priponskem drevesu in drugi verziji B–drevesa nizov so si podobni, vendar so pri slednji nekoliko boljši. Rezultati časovnih meritev so izraženi v sekundah.

število poizvedb	ERA	COSD	B-drevo nizov	
			v1	v2
5.000	0.06	0.30	0.04	0.05
10.000	0.12	0.83	0.08	0.10
15.000	0.19	0.88	0.12	0.14
20.000	0.23	1.24	0.16	0.18

Tabela 5.2: Rezultati časovnih meritev poizvedb.

5.3 Meritve V/I dostopov poizvedb

Obstajata dva modela računanja zahtevnosti V/I dostopov, splošen V/I model in predpomnilniško nezavedni model [7]. Razlikujeta se v tem, da sta v prvem modelu velikosti V/I bloka B in velikosti predpomnilnika M znani. Algoritmi, katerih delovanje temelji na teh vrednostih, so težje prenosljivi, saj se velikosti B in M lahko razlikujejo na drugih računalniških arhitekturah.

Algoritmem COSD ne predpostavi ničesar o velikosti B in M, zato sodi v predpomnilniški nezavedni model. Zahtevnost V/I dostopov pri iskanju niza P v strukturi COSD je vsota V/I dostopov iskanja niza po slepem drevesu in žirafi, to je $O(log_B|n| + |P|/B)$, kjer n predstavlja število zunanjih vozlišč številskega drevesa.

Rezultati meritev V/I dostopov poizvedb po priponskem drevesu, strukturi COSD in obeh verzijah B–drevesa nizov so prikazani v Tabelah: B.1, B.2, B.3 in B.4, podanih v Dodatku. Rezultati kažejo na to, da se vrednosti Ir, Dr, D1mr, DLmr, Dw in D1mw povečujejo za tolikokrat kot se poveča število poizvedb. Enako velja za I1mr, vendar samo za poizvedbe v priponskem drevesu in drugi verziji B–drevesa nizov. Vrednosti I1mr za strukturo COSD in prvo verzijo B–drevesa nizov ostajajo nespremenjene, ne glede na število poizvedb. Nespreminjajoče vrednosti ILmr in DLmw veljajo za vse strukture. Podane ugotovitve omogočajo razlago pridobljenih rezultatov za katerikoli testni primer, zato so v nadaljevanju predstavljeni rezultati meritev testnega primera s 5000 poizvedbami, za posamezno strukturo.

Vrednosti V/I dostopov smo delili s številom poizvedb, v našem primeru s 5000, in rezultate zaokrožili na celo število. Na ta način smo dobili podatke o tem,

koliko V/I dostopov je v povprečju potrebnih za posamezen iskani niz. Rezultate meritev V/I dostopov za posamezen iskani niz prikazuje Tabela 5.3.

	ERA	**COSD**	**B-drevo nizov**	
			v1	**v2**
Ir	50.067	62.692	58.997	59.062
I1mr	2	0	0	10
ILmr	0	0	0	0
Dr	12.693	12.016	21.448	21.429
D1mr	130	**1.913**	131	148
DLmr	59	**1.886**	6	5
Dw	45	144	204	214
D1mw	0	0	3	4
DLmw	0	0	0	0
LL	132	**1.914**	134	161

Tabela 5.3: Rezultati meritev V/I dostopov za posamezen iskani niz.

Vrednosti Ir, Dr in Dw predstavljajo vse možne V/I dostope do podatkov in ukazov programu, zato so tudi med najvišjimi. Iskanje niza po priponskem drevesu potrebuje najmanj ukaznih V/I dostopov (Ir), medtem ko jih največ porabi iskanje niza po strukturi COSD. Med obema verzijama B–drevesa nizov so vrednosti Ir razumljivo skoraj enaki. Bistvenih razlik med vrednostima Dr za iskanje niza v priponskem drevesu in strukturi COSD ni. Enako velja za obe verziji B–drevesa nizov. Iskanje v priponskem drevesu in strukturi COSD je dvakrat ceneje kot iskanje v obeh verzijah B–drevesa nizov. Vrednosti Dw so za vse strukture nizke, najnižja je pri iskanju v priponskem drevesu.

Vrednosti ILmr in DLmw so za vse strukture enake in so zanemarljivo majhne. Pri vrednostih I1mr izstopata druga verzija B–drevesa nizov in priponsko drevo, medtem ko pri vrednostih D1mw izstopata le obe verziji B–drevesa nizov.

Tabela 5.3 prikazuje, da iskanje niza v strukturi COSD ima največ zgrešenih bralnih V/I dostopov do podatkov na prvem in zadnjem nivoju predpomnilniške hierarhije, kljub temu, da je samo število branj skoraj identično številu branj pri priponskem drevesu. Medtem ko je vrednost D1mr za strukturo COSD v

primerjavi z ostalimi strukturami za 13 krat višja, je vrednost DLmr višja za kar 32 krat. Vrednost LL predstavlja vsoto vrednosti I1mr, D1mr in D1mw in je najvišja za strukturo COSD. Zanimivo je, da število zgrešenih dostopov do podatkov na zadnjem nivoju (DLmr), pri poizvedbi v strukturi COSD, predstavlja 99 odstotkov vseh zgrešenih dostopov, medtem ko je ta odstotek pri poizvedbi v priponskem drevesu dvakrat manjši.

Rezultati merjenj V/I dostopov, kažejo na to, da je iskanje niza v priponskem drevesu cenejše in bolj učinkovite v primerjavi z ostalimi strukturami. Sledita obe verziji B–drevesa nizov. Izkaže se, da je najbolj potratno iskanje nizov v strukturi COSD.

Ker imajo poizvedbe vseh testnih primerov v strukturi COSD najvišje število zgrešenih V/I dostopov do podatkov na zadnjem nivoju (DLmr), lahko pojasnimo vzrok, zakaj so se rezultati časovnih meritev poizvedb v strukturi COSD, prikazani v Tabeli 5.2, izkazali za najslabše. Namreč, na podlagi tehnične specifikacije [13] o glavnem pomnilniku strežnika, na katerem smo izvajali testiranje, lahko razberemo, da se en V/I dostop iz glavnega pomnilnika v predpomnilnik, izvede v 36 ns. Torej, če število zgrešenih V/I dostopov na zadnjem nivoju predpomnilniške hierarhije narašča, se povečujejo V/I dostopi v glavni pomnilnik in posledično povečuje čas izvajanja programa.

Poglavje 6

Sklepne ugotovitve

Primerjava in testiranje predstavljenih algoritmov je pokazala, da je algoritem ERA časovno zelo učinkovit, tako pri gradnji priponskega drevesa kot poizvedbah. Algoritem ERA postopek gradnje priponskega drevesa razdeli na več podproblemov, tako da gradi več manjših priponskih poddreves, jih lokalizira kar pomeni, da vsi potrebni podatki nahajajo v glavnem pomnilniku in tako zmanjša nepotrebne V/I dostope do podatkov. Za meritve se izkaže kot neprimeren, saj je čas iskanja enak vsoti potrebnega časa, da se podatki z diska prepišejo v pomnilnik ter časa poizvedb. Algoritem smo prilagodili, tako da se je priponsko drevo, v katerem so se iskali nizi, naložilo samo enkrat.

Ravno obratno velja za algoritem COSD, katerega gradnja podatkovne strukture je časovno dolgotrajen postopek, njena končna oblika shranjena na disku pa zavzame ogromno pomnilniškega prostora. Klub temu, da je izgradnja strukture COSD časovno in prostorsko potratna, kaže na nov koncept gradnje podatkovne strukture brez vedenja o velikosti posameznih nivojev pomnilniške hierarhije.

Prednost obeh verzij B–drevesa nizov, v primerjavi s priponskim drevesom in predvsem s strukturo COSD, je v časovno hitrejših poizvedbah. Slabost se izkaže v prostorski zahtevnosti.

Rezultati meritev V/I dostopov poizvedb kažejo, da je pri načrtovanju podatkovne strukture in organizacije podatkov potrebno upoštevati pomnilniško in predpomnilniško hierarhijo. V nasprotnem primeru se lahko soočimo s časovno neučinkovitimi poizvedbami.

Glede na pridobljene rezultate lahko zaključimo, da kljub uporabi žiraf in sle-

pih dreves, ki teoretično omejuje zahtevnost V/I dostopov poizvedb, za pripravljene testne primere, to ne drži. Razlog vidim predvsem v velikosti strukture COSD, saj v primerjavi z ostalimi podatkovnimi strukturami zasede daleč največ pomnilniškega prostora. Zaradi velikosti podatkovne strukture se večina podatkov nahaja v glavnem pomnilniku, zato je pri poizvedbah potrebno opraviti veliko V/I dostopov med posameznimi nivoji predpomnilniške hierarhije. Ne glede na to ali je implementirana strategija blokov učinkovita ali ne, slednja zaradi velikosti ne pride do izraza.

V tem diplomskem delu smo obravnavali samo prvi nivo pomnilniške hierarhije. Zanimivo bi bilo meritve ponoviti tudi za drugi nivo pomnilniške hierarhije (disk), kjer je čas dostopa še drastično večji in tudi velikost bloka B sorazmerno večja.

Če se v prihodnosti način implementacije uporabe žiraf izkaže tudi v praksi, bi bilo smiselno, za rešitev predlaganega izziva v uvodu, razmisliti o združitvi prednostih algoritmov ERA in COSD. Hibridni algoritem bi najprej, po konceptu algoritma ERA, zgradil priponsko drevo, ki ne bi predstavljalo končne oblike strukture. Končno obliko podatkovne strukture bi dobili v nadaljevanju, z učinkovitejšim načinom implementacije žiraf. Na koncu bi, za zapis nove podatkovne strukture na disk, uporabili koncept po *vanEmdeBoas*-u.

Dodatek A

Algoritmi

A.1 Vertikalna delitev

Algoritem 1: VerticalPartitioning

Input: Input string S, alphabet Σ, F_M

Output: Set of $VirtualTrees$

$VirtualTrees \leftarrow \emptyset$

$P \leftarrow \emptyset$ `// linked list of S-prefixes`

$P' \leftarrow$ **for** *every symbol* $s \in \Sigma$ **do** generate a S-prefix $p_i = s$

repeat

- scan input string S
- count in S the frequency f_{p_i} of every S-prefix $p_i \in P'$
- **forall the** $p_i \in P'$ **do**
  - **if** $0 < f_{p_i} \leq F_M$ **then** add p_i to P
  - **else forall the** *symbol* $s \in \Sigma$ **do** add $p_i s$ to P'
  - remove p_i from P'

until $P' = \emptyset$

sort P in descending f_{p_i} order

repeat

- $G \leftarrow \emptyset$ `// group of S-prefixes in a virtual tree`
- add $P.head$ to G and remove the item from P
- $curr \leftarrow$ next item in P
- **while** NOT *end of* P **do**
  - **if** $f_{curr} + SUM_{g_i \in G}(f_{g_i}) \leq F_M$ **then**
    - add $curr$ to G and remove the item from P
  - $curr \leftarrow$ next item in P
- add G to $VirtualTrees$

until $P = \emptyset$

return $VirtualTrees$

A.2 Horizontalna delitev - pripravljalna faza

Algoritem 2: HorizontalPartitioning.SubTreePrepare

Input: Input string S, S-prefix p

Output: Arrays **L** and **B** corresponding suffix sub-tree $\mathcal{T}_p$

L contains the locations of S-prefix p in string S

$\mathbf{B} \leftarrow \{\}$

$\mathbf{I} \leftarrow \{0, 1, ..., |\mathbf{L}| - 1\}$

$\mathbf{A} \leftarrow \{0, 0, ..., 0\}$

$\mathbf{R} \leftarrow \{\}$

$\mathbf{P} \leftarrow \{0, 1, ..., |\mathbf{L}| - 1\}$

$start \leftarrow |p|$ `// Start after S-prefix` p

while *there exists an undefined* $\mathbf{B}[i]$, $1 \leq i \leq |\mathbf{L}| - 1$ **do**

- $range \leftarrow GetRangeOfSymbols$ `// Elastic range`
- **for** $i \leftarrow 0$ *to* $|\mathbf{L}| - 1$ **do**
  - **if** $\mathbf{I}[i] \neq done$ **then**
    - $\mathbf{R}[\mathbf{I}[i]] \leftarrow ReadRange(S, \mathbf{L}[\mathbf{I}[i]] + start, range)$
    - // ReadRange(S,a,b) reads b symbols of S starting at position a
- **for** *every active area* AA **do**
  - Reorder the elements of **R**, **P** and **L** in AA so that **R** is lexicographically sorted. In the process maintain the index **I**
  - If two or more elements $\{a_1, ..., a_t\} \in AA, 2 \leq t$, exist such that $\mathbf{R}[a_1] = ... = \mathbf{R}[a_t]$ introduce for them a new active area
- **for** *all* i *such that* $\mathbf{B}[i]$ *is not defined,* $1 \leq i \leq |\mathbf{L}| - 1$ **do**
  - cs is the common prefix of $\mathbf{R}[i-1]$ and $\mathbf{R}[i]$
  - **if** $|cs| < range$ **then**
    - $\mathbf{B}[i] \leftarrow (\mathbf{R}[i-1][|cs|], \mathbf{R}[i][|cs|], start + |cs|)$
    - **if** $\mathbf{B}[i-1]$ *is defined or* $i = 1$ **then**
      - Mark $\mathbf{I}[\mathbf{P}[i-1]]$ and $\mathbf{A}[i-1]$ as *done*
    - **if** $\mathbf{B}[i+1]$ *is defined or* $i = |\mathbf{L}| - 1$ **then**
      - Mark $\mathbf{I}[\mathbf{P}[i]]$ and $\mathbf{A}[i]$ as *done* // last element of an active area
- $start \leftarrow start + range$

return (**L**,**B**)

A.3 Horizontalna delitev - gradnja priponskega drevesa

Algoritem 3: HorizontalPartitioning.BuildSubTree

Input: Arrays **L** and **B**

Output: The corresponding suffix sub-tree $\mathcal{T}_p$

$root \leftarrow$ new $Node(root)$

$u' \leftarrow$ new $Node$

$e' \leftarrow$ new $Edge(root, u')$

Label e' with $S_{L[0]}$ `// The suffix that corresponds L[0]`

Label u' with $\mathbf{L}[0]$ `// First (lexicographically) leaf`

Push e' to $Stack$

$depth \leftarrow |label(e')|$

for $i \leftarrow 1$ *to* $|\mathbf{B}| - 1$ **do**

- $(c_1, c_2, offset) \leftarrow \mathbf{B}[i]$
- **repeat**
  - Pop an edge $se(v_1, v_2)$ from the $Stack$ $depth \leftarrow depth - |label(se)|$
- **until** $depth \leq offset$
- **if** $depth = offset$ **then**
  - $u \leftarrow v_1$
- **else**
  - Break edge $se(v_1, v_2)$ into edges $se_1(v_1, v_t)$ and $se_1(v_t, v_2)$
  - Label se_1 with the first $offset$ symbols of $label(se)$
  - Label se_2 with the remaining symbols
  - $u \leftarrow v_t$
  - Push se_1 to $Stack$
  - $depth \leftarrow depth + |label(se_1)|$
- $u' \leftarrow$ new $Node$
- $ne \leftarrow$ new $Edge(u, u')$
- Label ne with $S_{L[i]}$ `// The suffix that corresponds L[i]`
- Label u' with $\mathbf{L}$[i] `// First (lexicographically) leaf`
- Push ne to $Stack$
- $depth \leftarrow depth + |label(ne)|$

return $\mathcal{T}_p$

A.4 Gradnja žiraf - Požrešna metoda

Algoritem 4: GreedyAlgorithm

Input: Trie T

Output: Set of Tries **G**

G $\leftarrow \{\}$

$n \leftarrow$ holds the number of leafs in T

$i \leftarrow 1$

while $i \leq n$ **do**

  $j \leftarrow i$

  while $j < n$ *and* $T^{i;j+1}$ *is giraffe tree* **do**

    $j \leftarrow j + 1$

  G.*add*($T^{i;j}$)

  $i \leftarrow j + 1$

return G

Dodatek B

Tabele

Priponsko drevo				
	5.000	**10.000**	**15.000**	**20.000**
Ir	250.337.290	513.329.064	765.802.961	1.011.846.959
I1mr	9.343	18.653	27.955	37.225
ILmr	30	30	30	30
Dr	63.465.975	130.134.220	194.215.306	256.408.146
D1mr	649.703	1.344.792	2.005.745	2.642.741
DLmr	296.007	607.061	903.608	1.195.154
Dw	225.669	451.151	675.974	900.801
D1mw	303	564	878	1.119
DLmw	0	0	1	0
LL	659.349	1.364.009	2.034.578	2.681.085

Tabela B.1: Rezultati meritev V/I dostopov poizvedb v priponskem drevesu.

Struktura COSD				
	5.000	**10.000**	**15.000**	**20.000**
Ir	313.462.342	643.159.322	960.435.328	1.266.625.394
I1mr	13	13	13	13
ILmr	13	13	13	13
Dr	60.081.292	123.255.741	184.063.639	242.758.988
D1mr	9.566.835	19.630.966	29.315.513	38.660.812
DLmr	9.429.168	19.302.277	28.881.620	38.008.358
Dw	722.296	1.446.712	2.169.920	2.893.089
D1mw	2.439	4.896	7.285	9.701
DLmw	0	0	0	0
LL	9.569.287	19.635.875	29.322.811	38.670.526

Tabela B.2: Rezultati meritev V/I dostopov poizvedb v strukturi COSD.

B–drevo nizov (v1)				
	5.000	**10.000**	**15.000**	**20.000**
Ir	294.983.321	604.713.816	899.075.500	1.187.300.402
I1mr	549	13	11	13
ILmr	11	13	11	13
Dr	107.241.582	219.910.969	326.927.927	431.676.663
D1mr	652.935	1.398.643	1.999.555	2.729.449
DLmr	27.674	54.089	81.036	107.306
Dw	1.021.729	2.366.352	3.543.632	4.726.067
D1mw	15.079	20.029	29.841	59.131
DLmw	0	0	0	0
LL	668.563	1.418.685	2.029.407	2.788.593

Tabela B.3: Rezultati meritev V/I dostopov poizvedb v B–drevesu nizov v1.

B–drevo nizov (v2)				
	5.000	**10.000**	**15.000**	**20.000**
Ir	295.310.698	605.368.396	900.057.672	1.188.610.095
I1mr	49.836	99.752	149.672	199.838
ILmr	11	13	11	16
Dr	107.144.055	220.055.901	327.145.371	431.966.618
D1mr	737.595	1.501.219	2.247.334	2.957.514
DLmr	24.316	47.464	70.705	93.420
Dw	1.071.729	2.436.287	3.648.584	4.866.015
D1mw	19.612	35.998	51.689	67.727
DLmw	0	0	0	0
LL	807.043	1.636.969	2.448.696	3.225.079

Tabela B.4: Rezultati meritev V/I dostopov poizvedb v B–drevesu nizov v2.

Literatura

[1] R. C. Deonier, S. Tavare, M. S. Waterman. *Computational Genome Analysis: An Introuduction*. Springer, 2005.

[2] P. Ferragina, R. Grossi. The string B–Tree: A New Data Structure for String Search in External MEmory and its Applications, *Journal of the ACM*, št. 46, str. 236–280, 1998

[3] S. Goldman, K. Goldman. *A Practical Guide to Data Structures and Algorithms Using Java*. Taylor & Francis Group, 2008.

[4] M. S. Goodrich, R. Tamassia. *Data Structures and Algorithms in Java, 4^{th} Edition*. John Wiley & Sons, Inc. 2004.

[5] E. Mansour, A. Allam, S. Skiadopoulos, P. Kalnis. ERA: Efficient Serial and Parallel Suffix Tree Construction for Very Long Strings, *Proceedings of the VLDB Endowment*, št. 1, zv. 5, str. 49–60, 2011.

[6] W. Stalling. *Operating Systems : Internals and Design Principles, 7^{th} Edition*. Prentice Hall, 2012.

[7] B. S. Carstensen, C. Torndahl. *Cache Oblivious String Dictionaries*. Magistrsko delo. Februar, 2007.

[8] I. Lipnik. *Iskanje v nestrukturiranih podatkih z uporabo B–dreves nizov*. Diplomsko delo. Junij, 2014

[9] DNK. Zadnji dostop, dne 01.07.2014, na: https://sl.wikipedia.org/wiki/Deoksiribonukleinska_kislina

[10] Memory close to the CPU: Caches. Zadnji dostop, dne 01.07.2014, na: http://db.inf.uni-tuebingen.de/files/teaching/ss09/dbcpu/dbms-cpu-6.pdf

[11] Programski paket Valgrind. Zadnji dostop, dne 01.07.2014, na: http://valgrind.org/

[12] Človeški genom. Zadnji dostop, dne 01.07.2014, na: http://cloud.kaust.edu.sa/Pages/Software.aspx

[13] Samsung Electronics. *Samsung 240pin Registered DIMM based on 2Gb D-die.* Samsung DDR3L SDRAM datasheet. Zadnji dostop, dne 01.07.2014, na: http://www.samsung.com/global/business/semiconductor/file/2011/-/product/2011/9/6/476443ds_ddr3_2gb_d-die_based_1_35v_rdimm_rev12.pdf

[14] Izvorne kode algoritmov, vhodni podatki in testni primeri poizvedb. Zadnji dostop, dne 07.07.2014, na: http://lusy.fri.uni-lj.si/sites/lusy.fri.uni-lj.si/files/publications/-/hmesic_thesis_data_20140707.zip